Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Staš Hvala

Metodologije testiranja programske opreme

DIPLOMSKO DELO
UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

prof. dr. Miha Mraz
MENTOR

doc. dr. Miha Moškon
SOMENTOR

Ljubljana, 2016

Tematika naloge:

Kandidat naj v svojem delu opravi pregled metodologij testiranja programske opreme. V nadaljevanju naj kandidat na vzorčni izvorni kodi preizkusi metodologiji enotskega in dinamičnega pomnilniškega testiranja s pomočjo aktualnih programskih orodij.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom prof. dr. Mihe Mraza in somentorstvom doc. dr. Mihe Moškona,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki "Dela FRI".

— Staš Hvala, Ljubljana, avgust 2016.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Staš Hvala

Metodologije testiranja programske opreme

POVZETEK

V pričujočem diplomskem delu predstavimo metode testiranja programske opreme in njihov pomen med in po razvojni fazi. Najprej skozi teoretičen pregled spoznamo vse glavne metode in nivoje testiranja ter opišemo njihovo uporabnost v realnem svetu.

V nadaljevanju zasnujemo lasten kratek program, ki ga testiramo s prosto dostopno programsko opremo. Program razbijemo na enote ter glede na njihovo funkcionalnost sestavimo testni scenarij in testne profile, ki služijo kot načrt za učinkovit testni postopek.

Za demonstracijo enotskega testiranja izberemo ogrodje *Google Test* in ga podrobneje opišemo. V okviru ogrodja sestavimo enotske teste, ki jih sproti poganjamo in z njimi preverjamo pravilnost delovanja programa.

Na koncu se osredotočimo še na potencialno puščanje pomnilnika v našem programu. V ta namen uporabimo orodje *Valgrind*, s katerim najprej testiramo naš program, nato pa z namensko okvaro kode preverimo še kvaliteto samega orodja.

Ključne besede: testiranje, programska oprema, enotski testi, Google Test, puščanje pomnilnika, Valgrind

University of Ljubljana
Faculty of Computer and Information Science

Staš Hvala

Investigation of software testing methodologies

ABSTRACT

In this thesis we investigate methodologies of software testing and emphasize their importance between and after the development phase. Firstly, we introduce the main methods and levels of testing and review their practicality in the real world applications.

Next we implement a short program, which is then tested with two different software tools. We break down the source code into units and, based on their functionality, devise a test strategy and test profiles which serve as a ground for an efficient test plan.

For the demonstration of unit testing we choose *Google Test* framework and illustrate its usage. With the help of the framework, we construct unit tests, which are simultaneously executed and used for checking whether the units they test are fit for use.

In the end we focus on the potential memory leaks in our program. For this reason, we use software development tool *Valgrind*, with which we first test our program and then assess the quality of the tool itself by intentionally harming the tested source code.

Key words: testing, software, unit tests, Google Test, memory leaks, Valgrind

ZAHVALA

Najprej bi se zahvalil mentorju prof. dr. Mihi Mrazu in somentorju doc. dr. Mihi Moškonu za odlično vodenje, hitre odzive in strpnost pri slovničnih napakah.

Hvala vsem sošolcem na fakulteti, ki so mi pomagali tekom študija.

Zahvala gre tudi moji družini in punci za dolgoletno podporo v dobrih in slabih časih študija.

Na koncu bi se zahvalil še razvijalcem orodja Valgrind in Google Test za njihovo odprtokodno rešitev.

— Staš Hvala, Ljubljana, avgust 2016.

KAZALO

Povzetek i

Abstract iii

Zahvala v

1 Uvod 1

1.1 Motivacija . . . 1

1.2 Cilji . . . 2

1.3 Metodologija . . . 2

2 Cilji in metode testiranja programske opreme 3

2.1 Cilji testiranja programske opreme . . . 3

2.2 Metode testiranja programske opreme . . . 4

2.2.1 Statično testiranje . . . 4

2.2.2 Dinamično testiranje . . . 5

2.2.3 Testiranje po metodi črne škatle . . . 6

2.2.4 Testiranje po metodi bele škatle . . . 8

2.3 Testni nivoji . . . 10

2.3.1 Enotsko testiranje . . . 11

2.3.2 Integracijsko testiranje . . . 12

2.3.3 Sistemsko testiranje . . . 13

2.3.4 Testiranje sprejetja . . . 13

2.4 Tipi testiranja . . . 13

3 Izbor programske opreme in testni profili 15

3.1 Testirana programska koda . . . 15

3.1.1 Enota *append* . . . 17
3.1.2 Enota *remove* . . . 17
3.1.3 Enota *print* . . . 17
3.1.4 Enota *sort* . . . 17
3.2 Načrt testiranja . . . 17
3.2.1 Testni scenarij . . . 18
3.2.2 Testni profili . . . 18

4 Realizacija testa in rezultati testiranja 21
4.1 Google Test . . . 21
4.1.1 Namestitev Google Testa . . . 22
4.1.2 Zagon testov . . . 23
4.1.3 Opis enotskih testov in rezultatov testiranja . . . 23
4.1.4 Ugotovitve . . . 27
4.2 Valgrind . . . 31
4.2.1 Zagon enotske kode z Valgrindom . . . 32
4.2.2 Odstranitev operatorja `delete` pri enoti *remove* . . . 34
4.2.3 Odstranitev destruktorja . . . 37
4.2.4 Uporaba ukaza `delete[]` namesto `delete` . . . 39
4.2.5 Uporaba ukaza `delete` na že izbrisanem vozlu . . . 41
4.2.6 Uporaba ukaza `delete` za neobstoječ vozel . . . 42
4.2.7 Ugotovitve . . . 43

5 Zaključek 45

1 Uvod

Testiranje je postalo ključna faza v razvoju programske opreme, zato je pomembno, da izberemo najustreznejši pristop glede na naravo samega problema. Namen te diplomske naloge je, da se poglobimo v teoretično ozadje testiranja in skozi praktične primere predstavimo njegovo relevantnost.

1.1 Motivacija

Želja vsakega programerja je venomer izdelovati popolno programsko opremo brez napak, ki zadovolji vse naročnikove specifikacije. Na žalost to ni mogoče že zaradi zmotljive narave človeka. Prav tako nikoli ne moremo biti prepričani v absolutno pravilnost programa. Možno pa je število napak močno minimizirati s temeljitim testiranjem. Drug problem predstavlja čas porabljen za samo testiranje, ki lahko ceno testiranja dvigne tudi do 40% razvojne cene produkta [1]. Za testiranje moramo torej uporabiti metodo, ki v najkrajšem možnem času odkrije največ napak oz. z največjo zagotovostjo potrdi pravilno delovanje produkta.

1.2 Cilji

Raziskali bi radi predvsem kakšne metodologije testiranja so danes na voljo, kdaj jih uporabiti, ter katere se najbolj obrestujejo glede na čas in zahtevnost testiranja. Med pestrim naborom orodij za testiranje bi radi izbrali tisto, ki bo v naši rešitvi odkrilo največ napak in odražalo najvišjo kvaliteto našega izdelka.

1.3 Metodologija

V diplomski nalogi uporabljamo metodološki pristop primerjalne študije, saj med seboj primerjamo različne načine testiranja programske opreme in orodij za odkrivanje napak.

V prvi fazi naredimo teoretični pregled čez vse vrste in načine testiranja. Izpostavimo kje nam posamezen pristop najbolj koristi, ter kakšne so njegove prednosti in slabosti.

V drugi fazi preidemo na praktičen del. Zasnujemo lasten program in s pomočjo izbranih orodij sistematično testiramo njegovo pravilnost. Skozi demonstracijo testiranja preverjamo tudi kvaliteto samih orodij in simuliramo realen razvoj in uporabo programske opreme.

2 Cilji in metode testiranja programske opreme

V pričujočem poglavju opravimo teoretičen pregled metodologij testiranja programske opreme. Na kratko predstavimo kakšne cilje si moramo zastaviti pred testiranjem, kakšne metode testiranja poznamo in kdaj jih uporabiti, ter skozi katere nivoje testiranja naj bi načeloma kvalitetno izdelana aplikacija morala iti.

2.1 Cilji testiranja programske opreme

Laično mišljenje je, da je testiranje namenjeno samo odkrivanju in preprečevanju programskih napak, a to opisuje samo **kratkoročne cilje** [2]. Odkrivanje programskih napak v fazi razvoja ima neposreden vpliv na njihovo preprečitev, saj se razvijalec uči iz svojih napak, kar eventualno pomeni manj programskih napak v poznejšem razvoju. **Dolgoročni cilji** so predvsem preverjanje kvalitete produkta in zadovoljstvo naročnika [2]. Na to večina programerjev pozabi, a ravno s temeljitim testiranjem naročniku dokažemo pravilno delovanje, kar pripomore pri končnemu zaupanju v izdelek. Pozabiti pa ne smemo na **cilje po končanem razvoju** [2], saj so s kvalitetno testirano programsko opremo posledično manjši tudi stroški vzdrževanja, ki jih želi vsak proizvajalec minimi-

zirati.

2.2 Metode testiranja programske opreme

Programsko opremo lahko testirajo tako razvijalci kot tudi naročniki oz. uporabniki, zato moramo metode testiranja že na začetku v grobem ločiti na več različnih skupin. Testirati je možno posamezne gradnike programske opreme ali pa kar celoten produkt. Poleg tega je pomembno še ali programsko opremo testiramo med delovanjem (dinamično testiranje), ali pred samim izvajanjem (statično testiranje). Razliko lahko opazimo že pri samem pisanju programa, kjer lahko napisano kodo statično analizira že večina današnjih integriranih razvojnih okolij (angl. *integrated development environment* - IDE). Dinamično analizo izvajamo sami kot razvijalci, ko zaganjamo program in preverjamo pravilnost njegovega delovanja.

2.2.1 Statično testiranje

Programsko opremo statično analiziramo brez izvajanja programske kode, kar močno omeji segmente, ki jih lahko pregledamo. Testiranje delimo na dva dela in sicer na ročni pregled kode in statično analizo. Prvi del opravijo razvijalci in po možnosti zunanji strokovnjaki. Statično analizo izvajamo s pomočjo raznih orodij, ki najpogosteje preverjajo sledeče:

- sintaktične napake,
- spremenljivke z nedefinirano vrednostjo,
- neuporabljene spremenljivke,
- nedosegljivo ali mrtvo kodo (angl. *dead code*),
- zastarele funkcije (angl. *deprecated function*),
- neupoštevanje standardov,
- varnostno ranljivost.

Orodja se po zmogljivosti zelo razlikujejo in so tesno odvisna od programskega jezika, v katerem je napisana programska oprema. Osnovno statično analizo izvajajo že standardna razvojna orodja, kot so prevajalniki (angl. *compilers*) in povezovalniki (angl. *linkers*). Pri statični analizi poznamo sledeče tehnike [3]:

- analiza toka programa (angl. *control flow analysis*),
- analiza pretoka podatkov (angl. *data flow analysis*),
- skladnost s standardi,
- izračun kodnih metrik.

Običajno se statično testiranje kombinira z metodo bele škatle (angl. *white-box*, glej razdelek 2.2.4) in se uporablja večinoma za večje in misijsko bolj kritične projekte, statično testiranje po metodi črne škatle (angl. *black-box*, glej razdelek 2.2.3) pa se v praksi uporablja za testiranje ali natančen pregled specifikacij [1].

Prednosti in slabosti statičnega testiranja so sledeče:

- Dobre lastnosti:
  - uporabi se lahko že v fazi razvoja, kar manjša stroške vzdrževanja,
  - potrebna je samo izvorna koda,
  - pregleda se lahko vse programske poti,
  - izvemo natančno lokacijo težave,
  - proces je relativno hiter, če uporabljamo ustrezna orodja.
- Slabe lastnosti:
  - gre za časovno zamudno opravilo, če ga izvajamo ročno,
  - večkrat spregleda napake,
  - ne odkrijemo napak, do katerih pride samo med izvajanjem.

2.2.2 Dinamično testiranje

Pri dinamičnem testiranju se osredotočimo na programsko opremo med izvajanjem (angl. *run time*). Poskušamo posnemati vsa možna stanja, ki jih lahko sistem zavzame in s samim izvajanjem izpostaviti spremenljivke, ki se spreminjajo skozi čas in so odvisne od prejšnjih iteracij. Da se lahko takšnega testiranja sploh poslužujemo, se mora program najprej sploh prevesti, torej se načeloma prej opravi statično testiranje (glej razdelek 2.2.1). Za dinamično testno okolje lahko poskrbimo sami, ali pa posežemo po raznih programih za dinamično testiranje. Ob uporabi zunanjih orodij je potrebno poudariti, da

lahko vplivajo na zmogljivost našega programa, kar je še posebej pomembno pri časovno občutljivih aplikacijah. Orodja za dinamično analizo nas obveščajo o stanju in obnašanju programa med samim izvajanjem. Informacije, ki jih običajno pridobimo iz tega pristopa, so težave z upravljanjem in puščanjem pomnilnika (angl. *memory leaks*), nepravilno upravljanje s kazalci (angl. *pointers*), analizo pokritosti (angl. *coverage analysis*) in analizo zmogljivosti (angl. *performance analysis*) [3]. Prednosti in slabosti dinamičnega testiranja so sledeče:

- Dobre lastnosti:
  - razkrije napake, ki jih s statično analizo ni moč odkriti ali pa so prekompleksne,
  - uporabnik prav tako uporablja program v dinamični obliki,
  - analizo lahko izvajamo brez izvorne kode,
  - preverimo lahko stvari, na katere nas je opozorila statična analiza.
- Slabe lastnosti:
  - časovno zamudnejše kot statična analiza,
  - težje je določiti lokacijo problema,
  - kakovost analize je močno odvisna od orodja, ki ga uporabljamo.

Pomembno je omeniti, da dinamičnega testiranja po metodi bele škatle neposredno ne enačimo z razhroščevanjem (angl. *debuggingom*), čeprav sta si na prvi pogled podobna. Cilj prvega je namreč identifikacija napake, cilj drugega pa njena odprava [1].

2.2.3 Testiranje po metodi črne škatle

Testiranje po metodi črne škatle bazira izključno na zahtevah in specifikacijah naročnika in nam v nasprotju s komplementom (glej razdelek 2.2.4) torej ni potrebno poznati internih poti, strukture ali implementacije [4]. Izraz črna škatla si lažje razložimo, če si za primer vzamemo računalnik, kjer mu z nekaterimi priključki dovajamo podatke (npr. miška, tipkovnica, tiskalnik itd.) z drugimi pa odvajamo (monitor, zvočniki itd.), a o sami realizaciji računalnika (v tem primeru ohišje računalnika dobesedno ponazarja črno škatlo) ne vemo ničesar in lahko priključene komponente testiramo samo tako, da preverjamo njihovo odzivnost (npr. premik miške na monitorju). Podobna zgodba je pri programski opremi, le da tukaj ne poznamo implementacije programske kode (razredov,

spremenljivk, funkcij, ...) pač pa lahko na podlagi vhodnih podatkov preverimo pravilnosti izhodnih. Običajen vrstni red procesa testiranja je sledeč [4]:

1. analiza zahtev in specifikacij,
2. izbira veljavnega ali neveljavnega vhoda glede na specifikacije,
3. določitev pričakovanega izhoda glede na vhod,
4. zasnova testov z izbranim vhodom,
5. izvedba definiranih testov,
6. primerjava dobljenih izhodov s pričakovanimi,
7. analiza rezultatov in odločitev o pravilni funkcionalosti produkta.

Tovrstno testiranje se lahko aplicira na vseh nivojih testiranja sistema - na enotskem (angl. *unit*), integracijskem (angl. *integration*), sistemskem (angl. *system*) in sprejemnem (angl. *acceptance*) nivoju (za obrazložitev testnih nivojev glej razdelek 2.3). Ob premikanju iz modula v podsistem in od tu v sistem se naša črna škatla veča in vhodi ter izhodi postajajo vse kompleksnejši, a se pristop k testiranju ohrani. Prav tako pa smo ob vse večjem in posledično tudi kompleksnejšem sistemu prisiljeni uporabljati metodo črne škatle, ker postane enostavno preveč internih poti, ki bi jih bilo potrebno stestirati po metodi testiranja bele škatle (glej razdelek 2.2.4) [4]. Seveda pa to ne pomeni, da je metoda črne škatle boljša. Problem se pojavi, ker oseba, ki izvaja teste (tester), nikoli ne more biti prepričana kdaj je bil produkt testiran na dovolj velikem nizu vhodov, da smo lahko prepričani o pravilnem delovanju, vse kombinacije pravilnih in nepravilnih vnosov pa je največkrat nemogoče preizkusiti.

Ključno vlogo pri metodi črne škatle igrajo testni primeri ali testni profili. Gre za vnaprej pripravljene množice vhodnih podatkov, s katerimi preverjamo pravilnost delovanja programske opreme [1]. Izvajalci testov se spet srečajo z dilemo o količini testov, ki jih je potrebno izvesti, da dosežemo dovolj visok procent zaupanja do pravilnosti delovanja produkta. Večina testov naj si bo podobnih, obvezno pa ne smemo pozabiti na testiranje skrajnih robnih primerov, ki so največkrat razlog hroščastega programa in jih lahko zlonameren uporabnik tudi izkoristi.

Izpostavimo še dobre in slabe lastnosti pristopa testiranja po metodi črne škatle [5]:

- Dobre lastnosti:
  - uporabno je za velike sisteme,
  - testi so ponovljivi (uporabno, ko nadgrajujemo sistem),
  - testiramo tudi okolje, v katerem bo programska oprema tekla,
  - zaradi neodvisnosti med testerjem in razvijalcam pride do objektivnega testiranja,
  - tester ne potrebuje posebnega tehničnega znanja in poznavanje implementacije sistema,
  - testi simulirajo uporabo sistema s strani navadnega uporabnika (angl. *end user's point of view*),
  - identificira nejasnosti in kontradikcije v specifikacijah,
  - testni profili so lahko zasnovani takoj, ko so postavljene specifikacije sistema.
- Slabe lastnosti:
  - razlog za napako v sistemu ostane neznan,
  - oteženo snovanje testnih profilov brez specifikacij,
  - problemi pri testiranju robnih primerov, če testni profili ne sledijo specifikaciji,
  - skoraj nemogoče je stestirati vse možne vhode v omejenem časovnem okvirju, zato je posledično pisanje profilov težavno in počasno delo,
  - med testnim procesom lahko nekaj programskih poti ostane nestestiranih,
  - problem je izpostavitvi kompleksnejši del sistema,
  - tester lahko preverja iste stvari, ki jih je preveril že razvijalec.

2.2.4 Testiranje po metodi bele škatle

Metoda bele škatle oz. strukturalno testiranje (angl. *structural testing*) je z razliko od metode črne škatle, ki testira funkcionalnost, testiranje strukturne narave. Imamo namreč vpogled v izvorno kodo, kar popolnoma spremeni način testiranja. Lahko se sicer spet skoncentriramo na vhode in izhode, v tem primeru pa lahko dodatno testiramo tudi posamezne module, poti, funkcije, zanke, pogoje ali stavke. Običajen vrstni red procesa testiranja je sledeč [4]:

1. analiza implementacije programske opreme,
2. identifikacija internih programskih poti,
3. izbira vhodov za preverjanje določenih poti,
4. izvedba definiranih testov,
5. primerjava dobljenih izhodov s pričakovanimi,
6. analiza rezultatov in odločitev o pravilni funkcionalosti produkta.

Običajno se to tehniko enači z enotskim testiranjem (angl. *unit testing*) (glej razdelek 2.3.1). Poudarek je torej na ločenih testih, a lahko zajema tudi testiranje povezanosti med posameznimi enotami programske opreme. Ponavadi je testiranje po metodi bele škatle časovno veliko bolj zamudno, a lahko ob skrbni izvedbi zagotovi odkritje večjega števila napak. Ta tip testiranja pogostejše izvajajo tudi razvijalci sami, saj že poznajo interno strukturo in lahko testirajo kodo kar med njenim pisanjem. Pri strukturnem testiranju je zelo pomembna informacija kakšno pokritost kode (angl. *code coverage*) smo dosegli, zato razdelimo testiranje na sledeče kriterije [6]:

- pokrivanje stavkov (angl. *statement coverage*),
- pokrivanje zank (angl. *loop coverage*),
- testiranje pogojev (angl. *branch coverage*),
- pokrivanje odločitev (angl. *decision coverage*),
- pokrivanje poti (angl. *path coverage*).

Vsaki programski opremi se ob takem testiranju v fazi razvoja določi željeno stopnjo pokritosti, kjer večja stopnja pomeni dražje (cenovno in časovno) ter kompleksnejše testiranje.

Na koncu izpostavimo še slabe in dobre lastnosti testiranja po metodi bele škatle [5]:

- Dobre lastnosti:
  - prisili razvijalca, da razmišlja o implementaciji programske opreme,
  - vodi do odkritja napak v kodi,
  - lahko privede do priložnosti za optimizacijo kode,

  - zaradi znanja o interni strukturi in implementaciji je lažje določiti vhod za efektivno testiranje,
  - možnost odkritja odvečnih vrstic kode.

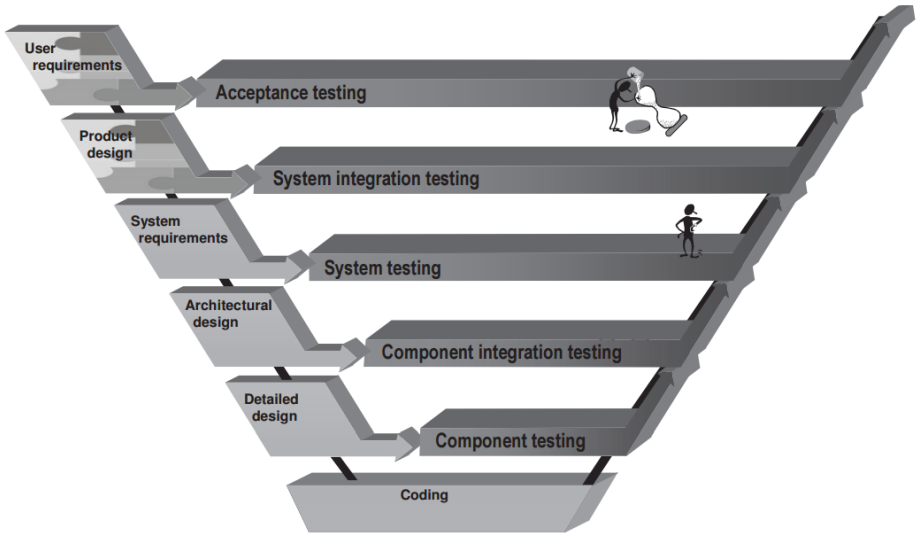
- Slabe lastnosti:
  - tovrstno testiranje je časovno zamudno,
  - tester potrebuje dodatno znanje in kompetence,
  - ni dovolj le statična analiza pač pa je potrebno kodo testirati tudi med izvajanjem,
  - za nekatere tipe testov je potrebno prilagajanje kode in vhodnih parametrov.

2.3 Testni nivoji

Na sliki 2.1 je prikazan *V-model* (angl. *Verification and Validation model*), ki služi kot dodatek k tradicionalnemu slapovnemu (angl. *Waterfall*) razvojnemu procesu. Razvojni cikel programske opreme razdeli na *verifikacijsko* (leva stran črke V) in *validacijsko* fazo (desna stran črke V). V verifikacijski fazi je predstavljenih več nivojev (število in točno definicijo nivojev prilagajamo glede na produkt), ki jih je potrebno izpolniti, da na koncu razvoja dobimo delujoč izdelek. Izpolnitev določenega nivoja verifikacijske faze preverimo z istoležečim nivojem validacijske faze na desni strani. V validacijski fazi imamo običajno sledeče dinamične testne nivoje [3]:

- enotsko testiranje (angl. *unit testing*),
- integracijsko testiranje (angl. *integration testing*),
- sistemsko testiranje (angl. *system testing*),
- testiranje sprejetja (angl. *acceptance testing*).

Testiranja se vršijo v istem zaporedju, kot so navedena. Pred vsakim začetkom testiranja predpostavimo, da se je na prejšnjem nivoju odpravilo vse napake. V naslednjih razdelkih podrobnejše opišem vse dinamične testne nivoje.



Slika 2.1 Primer izgleda *V-modela* [3].

2.3.1 Enotsko testiranje

Pri enotskem testiranju se je potrebno v razvojni ekipi najprej dogovoriti, kaj sploh je *enota*. Glede na zrnatost enot se seveda povečuje tudi število testiranj, ki jih je potrebno izvesti. Če je definicija enote preobsežna (npr. razred, angl. *class*) dobimo kompleksnejše rezultate in je vir napake težje odkriti, če pa je enota zelo majhna (npr. spremenljivka, angl. *variable*) pa iz rezultatov lahko ne dobimo nobenih uporabnih ugotovitev. Primer dobre enote v programski kodi je npr. funkcija, ki jo testiramo tako, da napišemo enotski test, ki funkcijo pokliče z določenimi parametri in preveri, če funkcija vrača pričakovan rezultat glede na vhod. Enote morajo biti torej enostavne, čim bolj neodvisne ter hitro izvedljive in razhrošč̌ljive. Cilj tega nivoja testiranja je, da se prepričamo o pravilnem delovanju posamičnih enot našega programa. Pri tem njihove povezanosti ne testiramo.

Prednost enotskega testiranja je v tem, da smo prisiljeni pisati modularno kodo in da se napisane teste lahko ponovno zaganja ob vsaki spremembi izvorne kode, kar služi kot grobo preverjanje porajanja novih napak. Dobra praksa je, da se enotske teste požene vsakič, kadar ponovno zgradimo (angl. *build*) program ali pa ga postavimo (angl. *deploy*) v produkcijsko okolje. Primer orodja, ki to podpira je Jenkins, odprtokodno orodje za

kontinuitetno integracijo [7].

2.3.2 Integracijsko testiranje

Pri integracijskem testiranju našo pozornost usmerimo na povezave med enotami (npr. objekti, moduli), kar je grafično predstavljeno na sliki 2.2. Ne glede na posamično pravilno funkcioniranje enot, je lahko izhod programa napačen, če pride do napake pri integraciji enot. Vsak program lahko načeloma napišemo v enem kosu, a to zelo oteži integracijsko testiranje, saj so prehodi med enotami manj očitni. Najboljša praksa je neodvisne dele programa čimbolj ločevati, kar pospeši odkrivanje napak na vmesnikih.

Pred izvajanjem testa najprej naredimo testni plan, ki med drugim pove tudi v kakšnem zaporedju bomo testirali. Poznamo štiri različne strategije pri zaporedju testiranja [3]:

- od zgoraj navzdol (angl. *top down*),
- od spodaj navzgor (angl. *bottom up*),
- funkcionalna integracija (angl. *functional integration*),
- veliki pok (angl. *big-bang*).

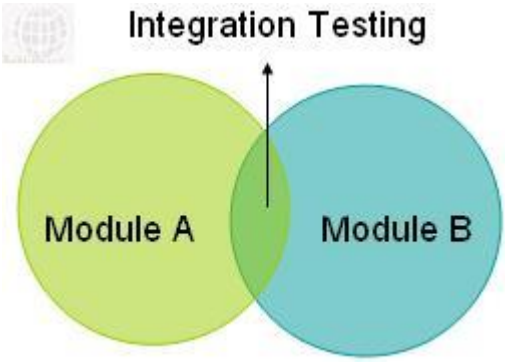
V *od zgoraj navzdol* integraciji začnemo testirati pri zgornjem vmesniku (npr. pri grafičnem uporabniškem vmesniku, angl. *general user interface* - GUI) in sledimo arhitekturni strukturi sistema. Pri tej strategiji uporabljamo tako imenovane testne nastavke (angl. *test stubs*), ki simulirajo neimplementirane module z vračanjem skladnih vrednosti. Prednost pristopa je, da testiramo po isti poti, ki ji bo kasneje sledil tudi uporabnik, slabost pa, da lahko celotno sistemsko pot testiramo šele po končanju produkta [8].

Kot ime pove, pri *od spodaj navzgor* integraciji začnemo s testiranjem pri spodnjen sloju. Tu se namesto *test stubs* za višje (neimplementirane) module kot nadomestek uporablja gonilnike (angl. *drivers*), ki kličejo (angl. *call*) implementirane module v fazi testiranja [3], [9]. Razvoj in testiranje potekata istočasno, za kompleksnejše programe pa moramo uporabiti večje število gonilnikov [8].

Pri *funkcionalni integraciji* združimo module glede na funkcionalnosti in skupke testiramo, kar je neka verzija od zgoraj navzdol strategije z vertikalno delitvijo.

Za *veliki pok* integracijo je značilno, da integriramo vse skupaj naenkrat in stestiramo izdelek v celoti. Je pa ob takem pristopu težje odkriti napake pri integraciji, ker testiramo

šele ob koncu razvoja. V praksi od zgoraj navzdol in od spodaj navzgor strategiji zaradi slabega planiranja ali izrednih situacij večkrat končata kot veliki pok pristop [3].



Slika 2.2 Grafični prikaz vsebine integracijskega testiranja [8].

2.3.3 Sistemsko testiranje

Sistemsko testiranje pride v poštev, ko končamo z razvojem aplikacije. Testiramo torej celoten integrirani sistem in preverimo, če se res ujema s specifikacijami naročnika. Logično lahko sklepamo, da bolj kot smo temeljito opravili *enotsko* in *integracijsko* testiranje, bolj efektivno je tudi samo sistemsko testiranje [3]. Pomembno je seveda tudi to, da imamo pred seboj vso dokumentacijo in specifikacijo sistema, ker le tako lahko ustvarimo učinkovit testni plan.

2.3.4 Testiranje sprejetja

To je zadnji nivo testiranja ob katerem naj bi bil prisoten tudi naročnik. V tej fazi se preverja ali je produkt že zrel za prehod v eksploatacijsko dobo [1]. Cilj tega testiranja ni, da najdemo napake kot pri ostalih nivojih, pač pa izvajamo nefunkcionalne teste s katerimi naročniku pokažemo izpolnitve vseh zahtev in kratko obrazložimo ubrane poti pri razvoju in delovanje samega produkta.

2.4 Tipi testiranja

Z različnimi tipi testiranj pokrijemo več možnosti za odpoved, ki jih z običajnimi pristopi ne bi odkrili. Sem spadajo tudi testi, ki obremenjujejo aplikacijo (angl. *stress tests*), in testiranje na širši množici uporabnikov (angl. *crowdsourced testing*). Seveda

za določene aplikacije ne potrebujemo izvesti vseh testiranj. Za sekvenči program nam tako ni potrebno izvajati testov za sočasnost (angl. *concurrent tests*).

V splošnem različni viri definirajo različne pristope k testiranju. Najtemeljitejše je delitev testiranja opisana v viru [10]. Slednji definira sledeče tipe testiranja:

- namestitveno testiranje (angl. *installation testing*),
- testiranje kompatibilnosti (angl. *compatibility testing*),
- dimno testiranje (angl. *smoke testing*),
- regresijsko testiranje (angl. *regression testing*),
- alfa testiranje (angl. *alpha testing*),
- beta testiranje (angl. *beta testing*),
- funkcionalno testiranje (angl. *functional testing*),
- kontinuitetno testiranje (angl. *continuous testing*),
- destruktivno testiranje (angl. *destructive testing*),
- zmogljivostno testiranje (angl. *performance testing*),
- testiranje uporabnosti (angl. *usability testing*),
- testiranje dostopnosti (angl. *accessibility testing*),
- testiranje varnosti (angl. *security testing*),
- testiranje internacionalizacije in lokalnosti (angl. *internationalization and localization*),
- razvojno testiranje (angl. *development testing*),
- A/B testiranje (angl. *A/B testing*),
- testiranje sočasnosti (angl. *concurrent testing*),
- testiranje skladnosti (angl. *conformance testing*).

3 Izbor programske opreme in testni profili

V pričujočem poglavju predstavimo uporabo enotskega testiranja na primeru vzorčne modularne programske kode. Zapišemo kratek testni scenarij ter testne profile, ki bodo preverili pravilnost napisanega programa.

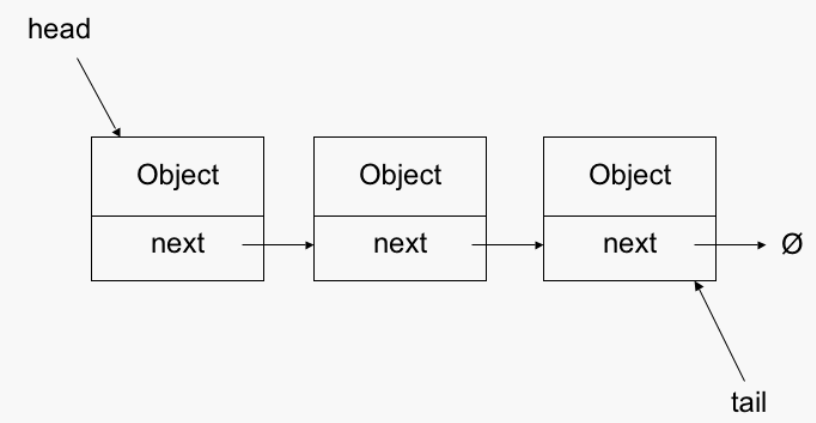
3.1 Testirana programska koda

Ob pregledu vseh načinov testiranj smo se odločili, da se osredotočimo podrobnejše na enotsko testiranje (glej razdelek 2.3.1) v kombinaciji z dinamičnim testiranjem po metodi črne škatle (glej razdelka 2.2.2 in 2.2.3) pristopom. Za enotsko testiranje smo se odločili, ker predvidevamo, da ima potencial v najkrajšem možnem času odkriti največ kritičnih napak in ker ima veliko prednost, da lahko vsako enoto testiramo že takoj, ko zaključimo z njenim pisanjem. Koncept testiranja po metodi črne škatle smo izbrali, ker je implementacija testov preprostejša in ker narava problema ne zahteva poznavanja interne strukture programa.

Obstaja pester nabor visokonivojskih programskih jezikov (angl. *high-level programming languages*) še več pa orodij oz. ogrodij (angl. *frameworks*), ki nudijo pomoč pri

razvoju programske opreme in izboljšujejo samo uporabniško izkušnjo jezika. Podobno je tudi pri ogrodjih za enotsko testiranje, a preden izbiramo ogrodje, se je modro najprej odločiti v katerem jeziku bomo sploh realizirali program, ker posledično olajša izbiro ogrodja. Izbrali smo jezik *C++*, saj poleg širokega nabora knjižnic ponuja tudi ročno upravljanje s pomnilnikom, kar predstavlja veliko prednost pri pisanju učinkovitih programov, a je hkrati lahko tudi glavni izvor težav.

Testiran program smo napisali sami, deluje pa kot enosmerni seznam (angl. *singly linked list*), ki uporabniku preko preprostega vmesnika (angl. *interface*) ponuja dodajanje, brisanje, izpis in urejanje elementov v seznamu. Izvorna koda je zastavljena čimbolj modularno, pomembne funkcije programa pa so razdeljene v enote. Seznam vsebuje vozle (angl. *nodes*), kjer vsak vozel v seznamu kaže na desnega soseda, zadnji vozel pa nima naslednika, zato kaže na ničelno vrednost (angl. *null*). Skica seznama je prikazana na sliki 3.1. Testirana koda se nahaja v elektronski prilogi.



Slika 3.1 Enosmerni seznam [11].

Kot smo že omenili, smo se odločili za razvoj vzorčnega programa v jeziku *C++*, zato smo za implementacijo seznama uporabil dve glavni programski paradigmi (angl. *programming paradigm*) tega jezika, objektno orientirano programiranje (angl. *object-oriented programming*, *OOP*) in kazalce (angl. *pointers*). Seznam se razdeli v dva razreda (angl. *class*). V prvemu razredu *Node* je realiziran vozel, ki mu ob instanciranju definiramo vrednost, ki jo drži. Vsakič, ko seznamu dodamo nov element, zadnjemu vozlu nastavimo kazalec na novega soseda. Drugi razred *LinkedList* vsebuje vse razpoložljive operacije seznama kot metode (angl. *methods*), ki sprejemajo ali pa vračajo neke vredno-

sti. Te operacije oz. metode predstavljajo tudi enote za testiranje. V sledečih razdelkih jih podrobnejše opišemo.

3.1.1 Enota *append*

V tej enoti seznamu pripenjamo elemente, ki jih je vnesel uporabnik. Tukaj ne pričakujemo kritičnih napak, ker se novi elementi dodajajo na konec seznama in do večjih težav ne more priti, se je pa vseeno pametno o tem dodatno prepričati s testiranjem.

3.1.2 Enota *remove*

Elemente iz seznama se lahko tudi odstrani. Za to poskrbi enota *remove*. Tukaj je možnih več potencialnih napak, saj je seznam lahko prazen, poln ali pa iskanega elementa sploh ni.

3.1.3 Enota *print*

Z enoto *print* iteriramo čez celoten seznam in izpišemo njegovo vsebino. Deluje samo za podatkovne tipe, ki lahko izpišejo svojo vsebino kot niz oz. implementirajo << operator za izpis vsebine. Z izpisom lahko uporabnik preveri tudi delovanje ostalih enot, zato si tukaj ne smemo privoščiti napačnih izpisov, kljub pravilnemu delovanju preostalega dela programa.

3.1.4 Enota *sort*

Enota *sort* uredi elemente seznama naraščajoče ali pa padajoče. Za delovanje je potrebno, da podatkovni tip podpira komparacijo med elementi istega tipa. Podatkovni tip mora torej vsebovati "je večji kot" (matematični simbol >) in "je manjši kot" (matematični simbol <) operator v svoji implementaciji.

3.2 Načrt testiranja

Pred implementacijo enotskih testov je potrebno definirati kaj je potrebno testirati in kako. Sestavimo torej dokument testiranja, ki koristi tako razvijalcu programske opreme kot testerju. Služi lahko tudi kot dokaz funkcionalnosti za naročnika.

3.2.1 Testni scenarij

Testni scenarij je dokument, ki z enovrstičnimi komentarji pove, kaj je potrebno testirati [12]. Ne opisuje postopkov testiranja, saj so temu namenjeni testni profili (glej razdelek 3.2.2), zato večkrat asociiramo testne profile s scenarijem. Testni profil torej opisuje *kako* testiramo, testni scenarij pa *kaj* testiramo [13]. Ironično je, da na spletu vsi avtorji člankov o testiranju poudarjajo kako pomemben je testni scenarij, a neke splošne recepture zanj ne obstaja. Najbrž zato, ker je njegov skelet močno vezan na naravo problema. Za enosmerni seznam smo glede na svoje enote sestavili sledeč scenarij:

1. testiraj dodajanje v seznam (enota *append*),
2. testiraj odstranjevanje iz seznama (enota *remove*),
3. testiraj izpisovanje vsebine seznama (enota *print*),
4. testiraj razčlenjevanje uporabnikovega vnosa (enota *sort*).

3.2.2 Testni profili

Testni profil je definiran kot dokument, ki določa testne vhode, pogoje za izvedbo in pričakovane rezultate za določene cilje, kot na primer izvedbo izbrane programske poti ali pa potrditev skladnosti s podanimi specifikacijami [14]. Testne profile sprva definira razvojna ekipa iz zahtev naročnika, poslovnih tveganj ali specifikacij programske opreme. S progresijo razvoja programske opreme postaja definicija testnih profilov vedno širša in bolj specifična [15]. Dobro zasnovani testni profili so sestavljeni iz vhodov, izhodov in vrstnega reda izvajanja [4]. Lahko jih izvajamo kaskadno (angl. *cascading*), torej zaporedno, ali pa neodvisno (angl. *independent*) [4]. Dokumentacija testnega profila naj bi vsebovala vsaj sledeče elemente [3]:

- unikatni identifikator (angl. *identifier* - ID),
- predpogoj za izvajanje,
- vhodne podatke in akcije,
- pričakovane rezultate,
- referenco na komponento, ki jo testiramo s tem testnim profilom.

Za demonstracijo enotskega testiranja je v našem primeru dodatna dokumentacija testnih profilov odvečna, ker je program obvladljiv (do 200 vrstic kode, angl. *line of code* - LOC) in je posledično za temeljito testiranje potrebno tudi manj testnih profilov. Za enosmerni seznam smo glede na testni scenarij postavili sledeče testne profile:

- dodajanje:
  - elementa v prazen seznam,
  - drugega elementa v seznam z enim elementom,
  - večjega števila elementov,
- odstranjevanje:
  - elementa iz seznama z enim elementom,
  - drugega elementa iz seznama z dvema elementoma,
  - prvega elementa iz seznama z dvema elementoma,
  - srednjega elementa iz seznama s tremi elementi,
  - večjega števila elementov iz polnega seznama,
  - elementa iz praznega seznama,
  - elementa, ki se ne nahaja v seznamu,
- izpisovanje:
  - seznama z enim elementom,
  - seznama z dvema elementoma,
  - seznama z večjim številom elementov,
  - praznega seznama,
- urejanje:
  - elemente naraščujoče,
  - elemente padajoče,
  - seznam z enim elementom,
  - prazen seznam.

Testne profile smo izvajali neodvisno, torej je vsak operiral na svojem seznamu. Pomemben je tudi vrstni red izvajanja. Predvsem se mora najprej potrditi pravilnost *dodajanja* v seznam, ker to enoto uporabljamo tudi pri ostalih testnih profilih.

4 Realizacija testa in rezultati testiranja

V pričujočem poglavju predstavimo orodje *Google Test*, ki smo ga uporabljali za realizacijo enotskih testov. Opišemo postopek namestitve orodja in podamo natančna navodila za zagon testov. Sledimo testnemu scenariju in s pomočjo orodja testne profile prevedemo v kodo za enotske teste. Zaradi neprepričljivih rezultatov uporabimo še orodje *Valgrind*, s katerim lahko dinamično poženemo program, orodje pa nas obvesti o potencialnih težavah s pomnilnikom.

4.1 Google Test

Jezik, v katerem smo napisali program, smo si že izbrali, kar posledično olajša izbiro orodja. Ob kratkem pregledu po svetovnem spletu smo ugotovili, da je trenutno najpopularnejšo ogrodje za enotsko testiranje *Google C++ Testing Framework* oz. neformalno *Google Test* ali *gtest* [16]. *Google Test* je *C++* ogrodje, ki omogoča lažje pisanje enotskih testov, deluje na vseh popularnih okoljih (angl. *cross-platform*) in bazira na *xUnit* arhitekturi, ki je *de facto standard* med ogrodji za enotsko testiranje ne glede na programski jezik, ki ga orodja dopolnjujejo [17]. Ponuja preprost in intuitiven programski vmesnik

(angl. *application programming interface*, *API*), kjer lahko z uporabo enega makroja (npr. `ASSERT_EQ`) hitro sestavljamo enotske teste.

4.1.1 Namestitev Google Testa

Google Test lahko namestimo na *Windows*, *OS X* ali *Linux* okolju [17]. Izbrali smo slednje, in sicer *Ubuntu 14.04 LTS* Linux distribucijo. Preden začnemo z namestitvijo moramo namestiti s pomočjo `apt-get install` (če uporabljate isto distribucijo) še pakete (angl. *packages*) `git`, `cmake` in `build-essential`.

Najprej iz *googlove github* strani https://github.com/google/googletest, na kateri se nahaja *Google Test*, kopiramo povezavo https://github.com/google/googletest.git. Nato zaženemo terminal in na željenem mestu v datotečnem sistemu uporabimo ukaz `git clone https://github.com/google/googletest.git`, ki klonira oddaljeno (angl. *remote*) *googletest* odlagališče (angl. *repository*). Avtomatsko se klonira oz. prenese zadnja stabilna verzija, v našem primeru *release-1.7.0*. Z ukazom za spremembo direktorija `cd` se pomaknemo v *googletest* imenik. Tam ustvarimo nov direktorij z `mkdir` (z imenom npr. *build*) in se pomaknemo vanj. Sedaj poženemo ukaz `cmake ..`, ki v predhodnem imeniku avtomatsko poišče datoteko *CMakeLists.txt* in generira ustrezen *MakeFile*. Pokličemo še `make`, ki zgradi (angl. *build*) oz. prevede (angl. *compile*) projekt glede na predhodno generiran *MakeFile*. Zaženemo `sudo cp -a ../include/gtest /usr/local/include`, da skopiramo *include* imenik in lahko *g++* povezovalnik (angl. *linker*) najde zaglavne datoteke (angl. *header files*) ogrodja. Za konec skopiramo še statične knjižnice (angl. *static library*) z ukazom `sudo cp -a libgtest.a libgtest_main.a /usr/local/lib`. Tako se ob prevajanju (angl. *compile time*) našega programa vsa potrebna koda za izvajanje testov iz teh dveh knjižnic poveže (angl. *links*) v naš program. Še enkrat povzemimo vse ukaze za terminal v zaporedju izvajanja v izpisu 4.1.

```
$ git clone https://github.com/google/googletest.git
$ cd googletest/googletest
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo cp -a ../include/gtest /usr/local/include
$ sudo cp -a libgtest.a libgtest_main.a /usr/local/lib
```

Izpis 4.1 Zaporedje ukazov za namestitev orodja Google Test.

4.1.2 Zagon testov

Pred zagonom testov moramo kodo najprej prevesti. Na tem mestu imamo dve opciji, in sicer lahko kodo prevedemo sami ali pa uporabimo `cmake` orodje na osnovi priložene *CMakeLists.txt* datoteke, ki smo jo napisali sami in vsebuje navodila za prevajanje programa. V obeh primerih se je pred začetkom potrebno s terminalom in `cd` ukazom premakniti v imenik, kjer se nahaja izvorna testna koda.

Za ročno prevajanje kode uporabimo ukaz `g++`. Temu dodamo vse datoteke, ki vsebujejo izvorno kodo programa in testov, torej *Node.h*, *LinkedList.h*, *unit_tests.cpp* in *main.cpp*. Dodamo stikalo `-o` in ime izhodne oz. izvršljive datoteke (angl. *executable file*). Na koncu povemo prevajalniku še, katere knjižnice bo potreboval. Dodamo torej še `-lgtest` in `-lpthread`, kjer je vrstni red pomemben. Celoten ukaz je prikazan v izpisu 4.2. Rezultat ukaza je izvršljiva datoteka, ki jo lahko poženemo z `./runtests`.

```
$ g++ Node.h LinkedList.h unit_tests.cpp main.cpp -o
  runtests -lgtest -lpthread
```

Izpis 4.2 Ukaz za ročno prevajanje enotskih testov.

Precej bolj preprosta opcija je uporaba `cmake` ukaza. Podobno, kot pri prevajanju orodja, najprej ustvarimo prazen direktorij z `mkdir` (z imenom npr. *build*) in se premaknemo vanj z `cd`. Poženemo `cmake ..` in nato `make`. Rezultat je isti, kot pri ročnem prevajanju. Tudi v tem primeru se namreč zgenerira izvršljiva datoteka. Zaporedje ukazov za *cmake* opcijo se nahaja v izpisu 4.3.

```
$ mkdir build
$ cd build
$ cmake ..
$ make
$ ./runtests
```

Izpis 4.3 Povzetek ukazov za prevajanje in zagon enotskih testov z orodjem *cmake*.

4.1.3 Opis enotskih testov in rezultatov testiranja

Za pomoč pri realizaciji testnih profilov v izbranem orodju se lahko poslužujemo dokumentacije in primerov izvedbe na *GoogleTest github* strani https://github.com/google/-googletest/blob/master/googletest/docs/Primer.md. Testne profile smo združili v skupine po enotah katerim pripadajo. Te so *testAppend*, *testRemove*, *testPrint* in *testSort*. Vsak testni profil smo deklarirali s pomočjo makroja `TEST(test_case_name,`

`test_name)`, kjer v prostor `test_case_name` vstavimo ime testne grupe, ki ji profil pripada (npr. *testAppend*), v prostor `test_name` pa unikatno ime testa, ki na kratko opiše kaj testira (npr. *appendElementToEmptyList*). V sledečih razdelkih predstavimo, kako smo realizirali teste in kakšen je bil rezultat izvedbe.

Dodajanje elementa v prazen seznam

Ustvarimo prazen seznam celih števil (angl. *Integers*) z `LinkedList<int> list`. Nato mu z `list.append(1)` pripnemo element in z `ASSERT_EQ(list[0], 1)` testiramo uspeh izvedbe. Test se je izvedel uspešno.

Dodajanje drugega elementa v seznam z enim

Ponovno ustvarimo prazen seznam s celimi števili, le da tukaj pripnemo dve števili. S tem testiramo, ali se seznam pravilno širi glede na dodane elemente. Z `ASSERT_EQ(list.size(), 2)` preverimo ali velikost seznama ustreza. Test se je izvedel uspešno.

Dodajanje večjega števila elementov

Ustvarimo prazen seznam primitivnega tipa (angl. *primitive types*), nato pa ga s `for` zanko napolnimo s poljubnimi vrednostmi tega tipa. Veliko število `ASSERT` makrojev se smatra za slabo prakso, zato gremo z drugo `for` zanko čez elemente ter preverjamo ustreznost. Ob primeru neujemanja postavimo zastavico `bool neq` na `true`. Po končanem iteriranju čez elemente testiramo stanje zastavice z `ASSERT_FALSE(neq)`. Test se je izvedel uspešno.

Odstrani element iz seznama z enim elementom

Ustvarimo prazen seznam, pripnemo element z `list.append(1)` in ga takoj zatem odstranimo z `list.remove(1)`. Testiramo, ali je seznam prazen z `ASSERT_EQ(list.size(), 0)`. Test se je izvedel uspešno.

Odstrani drugi element iz seznama dveh elementov

Ustvarimo prazen seznam, pripnemo dva elementa in odstranimo drugega. Ta test je potreben, da vidimo ali seznam pravilno odstranjuje vozel na katerega kaže glava seznama (angl. *head*). Testiramo, ali je velikost seznama enaka enemu elementu z `ASSERT_EQ(list.size(), 1)` in ali se neizbrisan element še vedno nahaja v seznamu z `ASSERT_EQ(list[0], 1)`. Test se je izvedel uspešno.

Odstrani prvi element iz seznama dveh elementov

Ustvarimo prazen seznam, pripnemo dva elementa in odstranimo prvega. S tem testom preverimo, ali se ob brisanju glave seznama drugi element pravilno prelevi v glavo seznama. Testiramo, ali je velikost seznama enaka enemu elementu z `ASSERT_EQ(list.size(), 1)` in ali smo dobili novo glavo seznama z `ASSERT_EQ(list[0], 2)`. Test se je izvedel uspešno.

Odstrani srednji element iz seznama treh elementov

Ustvarimo prazen seznam s celimi števili, pripnemo tri elemente in odstranimo srednjega. S tem preverimo, ali se ob brisanju srednjega elementa glava seznama pravilno poveže z zadnjim elementom. Testiramo, ali je velikost seznama enaka dvema elementoma z `ASSERT_EQ(list.size(), 2)` ter ali imamo v seznamu pravilna dva elementa z `ASSERT_EQ(list[0], 1)` in `ASSERT_EQ(list[1], 3)`. Test se je izvedel uspešno.

Odstrani večje število elementov

Ustvarimo prazen seznam s celimi števili in ga s `for` zanko napolnimo z vrednostmi od 0 do 99. Z naslednjo `for` zanko odstranimo vse dodane elemente in z `ASSERT_EQ(list.size(), 0)` testiramo velikost seznama. Test se je izvedel uspešno.

Odstrani iz praznega seznama

Ustvarimo prazen seznam s celimi števili in odstranimo poljuben element, čeprav je seznam prazen. Z `ASSERT_EQ(list.size(), 0)` preverimo, da se velikost seznama ni spremenila. Test se je sprva izvedel neuspešno, program in njegovo testiranje sta se neuspešno zaključila, operacijski sistem pa nam je vrnil napako *segmentation fault*, kar pomeni, da smo posegali po nedovoljeni lokaciji v pomnilniku. Ob ponovnem pregledu implementacije seznama smo ugotovili, da je program iskal vozlišče z določeno vrednostjo, tudi če je bil seznam prazen. To napako smo hitro odpravili s predhodnim preverjanjem velikosti seznama pred samim odstranjevanjem. V primeru odstranjevanja iz praznega seznama vrnemo izjemo (angl. *exception*), ki obvesti uporabnika, da je operacija ilegalna. V testnem profilu smo `ASSERT_EQ(list.size(), 0)` nadomestili z `ASSERT_THROW(list.remove(1), std::out_of_range)`, ki pričakuje izjemo `out_of_range` ob odstranjevanju elementa iz praznega seznama. Velja omeniti, da če ne

uporabimo `ASSERT_THROW` v testu, nas ogrodje vseeno obvesti, da je ujelo izjemo in označi test kot *neuspešen* (angl. *failed*), kot je to vidno na poročilu v izpisu 4.5.

Odstrani element, ki ne obstaja

Ustvarimo prazen seznam s celimi števili, ga napolnimo z naključnimi števili in odstranimo število, ki v seznamu ne obstaja. Z `ASSERT_EQ(list.size(), 3)` testiramo, ali se velikost seznama ohrani. Pri testu je bil rezultat podoben kot pri prejšnjemu testu. Dobili smo signal *SEGFAULT* in program se je zaključil. Program smo popravili tako, da smo uvedli vračanje izjeme v primeru, da je dosežen konec seznama, ker elementa nismo našli. Tudi tukaj nadomestimo prejšnji makro z `ASSERT_THROW(list.remove(3), std::out_of_range)`.

Izpiši seznam z enim elementom

Ustvarimo prazen seznam s celimi števili in mu pripnemo en element, z `ASSERT_EQ(list.print(), "1")` pa testiramo pravilnost izpisa. Test se je izvedel uspešno.

Izpiši seznam z dvema elementoma

Ustvarimo prazen seznam s celimi števili in mu pripnemo dva elementa. Tukaj pričakujemo, da bo seznam elementa med seboj tudi jasno ločil, v našem primeru s puščico $->$. Ločilni znak pa se ne sme pojaviti na koncu seznama. Pravilnost izpisa testiramo z `ASSERT_EQ(list.print(), "1 -> 2")`. Test se je izvedel uspešno.

Izpiši seznam z večjim številom elementov

Ustvarimo prazen seznam s celimi števili in ga z `for` zanko napolnimo z vrednostmi od 0 do 99, sočasno pa v `std::ostringstream os` pripenjamo pričakovan izpis za kasnejše preverjanje. Z `ASSERT_EQ(list.print(), os.str())` testiramo pravilnost izpisa. Test se je izvedel uspešno.

Izpiši prazen seznam

Ustvarimo prazen seznam s celimi števili in ga izpišemo. Zaradi napak pri prejšnjih testnih profilih, ki so testirali prazen seznam, smo bili tukaj previdni in že takoj na začetku enote v primeru praznega seznama vrnemo prazen niz. Izpis testiramo z `ASSERT_EQ(list.print(), "")`. Test se je izvedel uspešno.

Uredi seznam naraščujoče

Ustvarimo prazen seznam s celimi števili in ga s `for` zanko napolnimo z vrednostmi od 99 do 0. Sortiramo seznam z `list.sort(true)`, kjer prek parametra z `bool` zastavico nastavimo, da želimo naraščujoče zaporedje. Naredimo še eno `for` zanko in preverjamo, če urejenost elementov ustreza. V primeru neustrezanja postavimo zastavico `bool neq` na `true`. Na koncu testiramo stanje zastavice z `ASSERT_FALSE(neq)`. Test se je izvedel uspešno.

Uredi seznam padajoče

Pri tem testu napolnimo seznam z vrednostmi v obratnem vrstnem redu kot pri prejšnjem testu, torej od 0 do 99. Za sortiranje uporabimo `list.sort(false)`, da nastavimo padajoče zaporedje. Test se je izvedel uspešno.

Uredi seznam z enim elementom

Ustvarimo prazen seznam s celimi števili in mu pripnemo en element. Seznam sortiramo, a klic metode prestavimo v `ASSERT_NO_THROW(list.sort())`. Tako ogrodju eksplicitno povemo, da ne želimo, da pride do izjeme. Z `ASSERT_EQ(list[0], 1)` dodatno testiramo še, če v seznamu ni prišlo do sprememb in če vsebuje samo en element. Test se je izvedel uspešno.

Uredi prazen seznam

Ustvarimo prazen seznam s celimi števili in ga sortiramo. Tukaj smo bili pri implementaciji ponovno previdni in smo v primeru sortiranja praznega seznama vrnili `std::out_of_range` izjemo. To testiramo z `ASSERT_THROW(list.sort(), std::out_of_range)`. Test se je izvedel uspešno.

4.1.4 Ugotovitve

Pognali smo skupno osemnajst testov. Iz poročila v izpisu 4.4 lahko razberemo, da so se vsi testi izvedli uspešno. Orodje razdeli izpis na štiri testne primere (angl. *test cases*), pove koliko testov se je pognalo in kakšen je njihov rezultat. Zeleni *OK* napis predstavlja uspešen izid, medtem, ko bi rdeči *FAILED* napis predstavljal neuspešen izid. Poleg rezultata izvedbe testov nam orodje izpiše še metriko časovnega izvajanja testa v

milisekundah. V našem primeru so se vsi testi izvedli v *0 ms*, ker je program kratek in testi niso procesorsko zahtevni.

```
[==========] Running 18 tests from 4 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from testAppend
[ RUN      ] testAppend.appendElementToEmptyList
[       OK ] testAppend.appendElementToEmptyList (0 ms)
[ RUN      ] testAppend.appendSecondElement
[       OK ] testAppend.appendSecondElement (0 ms)
[ RUN      ] testAppend.appendMultipleElements
[       OK ] testAppend.appendMultipleElements (0 ms)
[----------] 3 tests from testAppend (0 ms total)

[----------] 7 tests from testRemove
[ RUN      ] testRemove.removeFromOneElementList
[       OK ] testRemove.removeFromOneElementList (0 ms)
[ RUN      ] testRemove.removeSecondFromTwoElementList
[       OK ] testRemove.removeSecondFromTwoElementList (0 ms)
[ RUN      ] testRemove.removeFirstFromTwoElementList
[       OK ] testRemove.removeFirstFromTwoElementList (0 ms)
[ RUN      ] testRemove.removeMiddleFromThreeElementList
[       OK ] testRemove.removeMiddleFromThreeElementList (0 ms)
[ RUN      ] testRemove.removeMultipleElements
[       OK ] testRemove.removeMultipleElements (0 ms)
[ RUN      ] testRemove.removeFromEmptyList
[       OK ] testRemove.removeFromEmptyList (0 ms)
[ RUN      ] testRemove.removeNonexistentElement
[       OK ] testRemove.removeNonexistentElement (0 ms)
[----------] 7 tests from testRemove (0 ms total)

[----------] 4 tests from testPrint
[ RUN      ] testPrint.printOneElementList
[       OK ] testPrint.printOneElementList (0 ms)
[ RUN      ] testPrint.printTwoElementList
[       OK ] testPrint.printTwoElementList (0 ms)
[ RUN      ] testPrint.printMultipleElementList
[       OK ] testPrint.printMultipleElementList (0 ms)
[ RUN      ] testPrint.printEmptyList
[       OK ] testPrint.printEmptyList (0 ms)
[----------] 4 tests from testPrint (0 ms total)

[----------] 4 tests from testSort
[ RUN      ] testSort.sortElementListAscending
[       OK ] testSort.sortElementListAscending (0 ms)
[ RUN      ] testSort.sortElementListDescending
[       OK ] testSort.sortElementListDescending (0 ms)
[ RUN      ] testSort.sortOneElementList
[       OK ] testSort.sortOneElementList (0 ms)
```

```
[ RUN      ] testSort.sortEmptyList
[       OK ] testSort.sortEmptyList (0 ms)
[----------] 4 tests from testSort (0 ms total)

[----------] Global test environment tear-down
[==========] 18 tests from 4 test cases ran. (0 ms total)
[  PASSED  ] 18 tests.
```

Izpis 4.4 Poročilo uspešne izvedbe naših enotskih testov.

```
[==========] Running 18 tests from 4 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from testAppend
[ RUN      ] testAppend.appendElementToEmptyList
[       OK ] testAppend.appendElementToEmptyList (0 ms)
[ RUN      ] testAppend.appendSecondElement
[       OK ] testAppend.appendSecondElement (0 ms)
[ RUN      ] testAppend.appendMultipleElements
[       OK ] testAppend.appendMultipleElements (0 ms)
[----------] 3 tests from testAppend (0 ms total)

[----------] 7 tests from testRemove
[ RUN      ] testRemove.removeFromOneElementList
[       OK ] testRemove.removeFromOneElementList (0 ms)
[ RUN      ] testRemove.removeSecondFromTwoElementList
[       OK ] testRemove.removeSecondFromTwoElementList (0 ms)
[ RUN      ] testRemove.removeFirstFromTwoElementList
[       OK ] testRemove.removeFirstFromTwoElementList (0 ms)
[ RUN      ] testRemove.removeMiddleFromThreeElementList
[       OK ] testRemove.removeMiddleFromThreeElementList (0 ms)
[ RUN      ] testRemove.removeMultipleElements
[       OK ] testRemove.removeMultipleElements (0 ms)
[ RUN      ] testRemove.removeFromEmptyList
unknown file: Failure
C++ exception with description "LinkedList::remove: List is empty!"
thrown in the test body.
[  FAILED  ] testRemove.removeFromEmptyList (0 ms)
[ RUN      ] testRemove.removeNonexistentElement
[       OK ] testRemove.removeNonexistentElement (0 ms)
[----------] 7 tests from testRemove (1 ms total)

[----------] 4 tests from testPrint
[ RUN      ] testPrint.printOneElementList
[       OK ] testPrint.printOneElementList (0 ms)
[ RUN      ] testPrint.printTwoElementList
[       OK ] testPrint.printTwoElementList (0 ms)
[ RUN      ] testPrint.printMultipleElementList
[       OK ] testPrint.printMultipleElementList (0 ms)
[ RUN      ] testPrint.printEmptyList
[       OK ] testPrint.printEmptyList (0 ms)
```

```
[----------] 4 tests from testPrint (0 ms total)

[----------] 4 tests from testSort
[ RUN      ] testSort.sortElementListAscending
[       OK ] testSort.sortElementListAscending (0 ms)
[ RUN      ] testSort.sortElementListDescending
[       OK ] testSort.sortElementListDescending (0 ms)
[ RUN      ] testSort.sortOneElementList
[       OK ] testSort.sortOneElementList (0 ms)
[ RUN      ] testSort.sortEmptyList
[       OK ] testSort.sortEmptyList (0 ms)
[----------] 4 tests from testSort (0 ms total)

[----------] Global test environment tear-down
[==========] 18 tests from 4 test cases ran. (1 ms total)
[  PASSED  ] 17 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] testRemove.removeFromEmptyList

 1 FAILED TEST
```

Izpis 4.5 Primer neuspešnega poročila za test "removeFromEmptyList" zaradi neobravnavane izjeme.

Potrdili smo osnovno pravilnost delovanja predstavljenih enot, a še vedno nismo popolnoma prepričani v brezhibno delovanje programa. Pred samim pisanjem enotskih testov smo pričakovali, da bo orodje bolj informativno in omogočalo fleksibilnejše preverjanje kode. Tudi v dveh primerih, ko smo imeli napako v programu, je orodje prenehalo delovati in ni ponudilo dodatnega izpisa zaradi narave pomniliških napak. Smo pa spoznali, kakšna je največja prednost ogrodij za enotsko testiranje. Kot smo že omenili, se enotske teste piše vzporedno z razvijanjem programa, kar ne potrdi le delovanja enot, pač pa nas prisili, da razmislimo in odkrijemo morebitne robne pogoje, za katere smo pozabili implementirati izjemo, ali pa smo preprosto naredili napako v realizaciji. Izjemna prednost je tudi ponovljivost izvedbe testov. Vsakič, ko v izvorni kodi naredimo še tako malenkostno spremembo, lahko ponovno poženemo enotske teste in preverimo, če program še vedno deluje brez napak.

Velika pomanjkljivost *Google Test* in vseh ostalih *C++* ogrodij za enotsko testiranje je preverjanje puščanja pomnilnika. Programska jezika *C* in *C++* slovita po svoji hitrosti, ki izhaja tudi iz ročnega upravljanja s pomnilnikom. Pri jezikih kot je npr. *Java*, za avtomatsko sproščanje pomnilnika poskrbi proces *garbage collection*. Če se že poslužujemo funkcijam za upravljanje s pomnilnikom, je potrebno tudi ustrezno testiranje pravilnega sproščanja pomnilnika. Tudi če se program uspešno izvede, lahko brez opozo-

rila za seboj pusti sledi nesproščenega pomnilnika. Program torej po nepotrebnem zaseda pomnilniške vire in tako potencialno upočasni celoten sistem. Naš program prav tako vsebuje sistemske klice za alokacijo in dealokacijo pomnilnika, zato sledi logičen sklep, da bi radi preverili tudi kaj se dogaja z rezerviranim pomnilnikom. Ker nam ogrodja za enotsko testiranje ne ponujajo tega testiranja, smo prisiljeni, da poiščemo alternativno orodje, ki je po možnosti eksplicitno namenjeno pomnilniškemu testiranju.

4.2 Valgrind

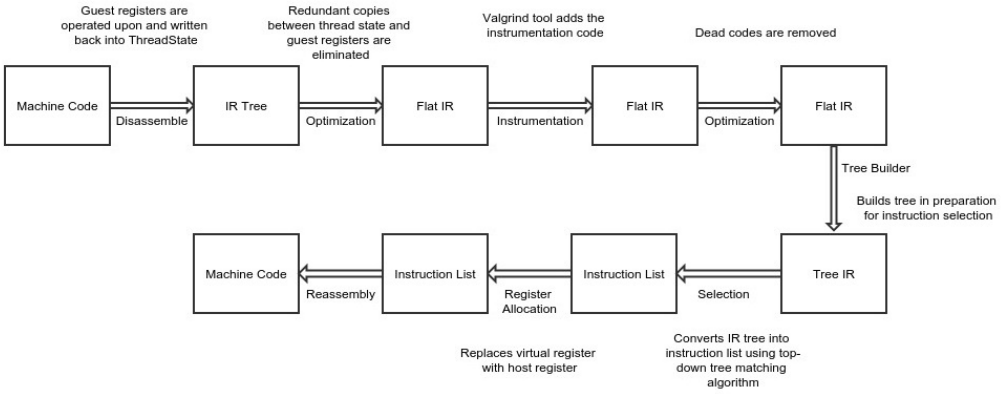
Valgrind je odprtokodno (angl. *open source*) in brezplačno instrumentacijsko ogrodje za izgradnjo orodij za dinamično analizo. Podporo nudi za kar nekaj platform, med drugimi tudi *Linux* in *Android*. V okviru *Valgrind* distribucije nam razvijalci ponujajo šest različnih orodij primernih za produkcijo in sicer *memory error detector*, *two thread error detectors*, *cache and branch-prediction profiler*, *call-graph generating cache and branch-prediction profiler* in *heap profiler* [18]. Za nas je najbolj zanimiv prvi, torej detektor za pomnilniške napake, znan tudi pod imenom *memcheck*. To orodje nam omogoča detekcijo sledečih pogostih napak v *C* in *C++* programih [19]:

- dostopanja do nedovoljenih lokacij v pomnilniku,
- uporabe nedefiniranih vrednosti,
- nepravilnega sproščanja pomnilnika,
- prekrivanja kazalcev v `memcpy` funkciji,
- podajanja sumljivih vrednosti funkcijam za alokacijo pomnilnika,
- puščanja pomnilnika.

Valgrind je izredno preprosto namestiti. Če se vrnemo na našo *Linux* distribucijo *Ubuntu 14.04 LTS*, v njej najprej zaženemo terminal in vpišemo `sudo apt-get install valgrind`. Orodje se avtomatsko prenese in namesti ter je pripravljeno za uporabo. Alternativno ga lahko prenesemo ročno iz njihove spletne strani http://valgrind.org-/downloads/current.html in prevedemo z uporabo ukazov `make` in `make install`.

Delovanje *Valgrinda* je istočasno prednost in slabost. Orodja nam namreč ni potrebno vključiti pri prevajanju programa, ampak ga uporabimo šele ob izvajanju. Razlog tiči

v implementaciji orodja. Ta uporablja prevajanje med zagonom programa (angl. *just-in-time*, JIT) in deluje kot posrednik med operacijskim sistemom in našim programom. Ustvari namreč sintetično centralno procesno enoto (angl. *central processing unit*, CPU), ki posebej sprocesira naš program. Preden *Valgrind* jedro ukaze izvede, jih preda še izbranim orodjem (npr. *memcheck*), da ta dodajo svojo instrumentacijsko kodo (angl. *instrumentation code*) za prestrezanje sistemskih klicov (angl. *system calls*) in beleženje stanja sistema pred in po izvedbi klicov. Virtualno jedro nato zažene modificirano kodo in izpiše vse napake in obvestila o morebitnem puščanju pomnilnika preden kontrolo preda pravemu procesorju [20]. Podrobnejši prikaz delovanja je viden na sliki 4.1. Slabost *Valgrinda* je zaradi dodatne kode močna obremenitev sistema in občutno počasnejše izvajanje našega programa. Orodje torej ni namenjeno za aplikacije, kjer je zmogljivost kritičnega pomena.



Slika 4.1 Diagram delovanja orodja *Valgrind* [21].

4.2.1 Zagon enotske kode z Valgrindom

Za zagon enotskih testov se najprej v terminalu z ukazom `cd` premaknemo v direktorij, kjer smo prevedli našo kodo (npr. *build*). Klasičnemu zagonu izvršljive datoteke z `./` spredaj dodamo še ukaz `valgrind` in ustrezne parametre. Ukaz za zagon je viden v izpisu 4.6. S stikalom `--tool` izberemo orodje *memcheck*, `--leak-check=full` pa nam da podrobnejšo informacijo o vsakem izgubljenem bloku v pomnilniku [19]. Program se začne z izpisom osnovnih informacij o orodju, kjer številka znotraj dveh == predstavlja identifikacijsko številko procesa (angl. *process identifier*, PID), nato pa se izpišejo

rezultati enotskih testov na enak način kot prej. Na koncu izpisa našega programa se pod *HEAP SUMMARY* pojavijo rezultati o stanju pomnilnika. Iz poročila v izpisu 4.7 je razvidno, da smo ob izvajanju našega programa alocirali in sprostili pomnilnik 1.114 krat in alocirali 98.212 bajtov spomina. Iz teh podatkov in obvestila *All heap blocks were freed – no leaks are possible* je možno razbrati, da pri našem programu ne prihaja do težav s pomnilnikom. Je pa možno iz časovnih metrik *Google Test* orodja razbrati za koliko *Valgrind* upočasni testiran program, kajti čas izvajanja vseh testov skupaj se je povečal iz *0 ms* na *134 ms*.

```
    $ valgrind --tool==memcheck --leak-check=full ./runtests
```

Izpis 4.6 Ukaz za zagon enotskih testov v okviru *Valgrind* orodja.

```
==12639== Memcheck, a memory error detector
==12639== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==12639== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==12639== Command: ./runtests
==12639==
[==========] Running 18 tests from 4 test cases.
[----------] Global test environment set-up.
[----------] 3 tests from testAppend
[ RUN      ] testAppend.appendElementToEmptyList
[       OK ] testAppend.appendElementToEmptyList (9 ms)
[ RUN      ] testAppend.appendSecondElement
[       OK ] testAppend.appendSecondElement (1 ms)
[ RUN      ] testAppend.appendMultipleElements
[       OK ] testAppend.appendMultipleElements (2 ms)
[----------] 3 tests from testAppend (23 ms total)

[----------] 7 tests from testRemove
[ RUN      ] testRemove.removeFromOneElementList
[       OK ] testRemove.removeFromOneElementList (1 ms)
[ RUN      ] testRemove.removeSecondFromTwoElementList
[       OK ] testRemove.removeSecondFromTwoElementList (2 ms)
[ RUN      ] testRemove.removeFirstFromTwoElementList
[       OK ] testRemove.removeFirstFromTwoElementList (1 ms)
[ RUN      ] testRemove.removeMiddleFromThreeElementList
[       OK ] testRemove.removeMiddleFromThreeElementList (2 ms)
[ RUN      ] testRemove.removeMultipleElements
[       OK ] testRemove.removeMultipleElements (2 ms)
[ RUN      ] testRemove.removeFromEmptyList
[       OK ] testRemove.removeFromEmptyList (33 ms)
[ RUN      ] testRemove.removeNonexistentElement
[       OK ] testRemove.removeNonexistentElement (2 ms)
[----------] 7 tests from testRemove (44 ms total)

[----------] 4 tests from testPrint
```

```
[ RUN      ] testPrint.printOneElementList
[       OK ] testPrint.printOneElementList (3 ms)
[ RUN      ] testPrint.printTwoElementList
[       OK ] testPrint.printTwoElementList (3 ms)
[ RUN      ] testPrint.printMultipleElementList
[       OK ] testPrint.printMultipleElementList (7 ms)
[ RUN      ] testPrint.printEmptyList
[       OK ] testPrint.printEmptyList (1 ms)
[----------] 4 tests from testPrint (14 ms total)

[----------] 4 tests from testSort
[ RUN      ] testSort.sortElementListAscending
[       OK ] testSort.sortElementListAscending (2 ms)
[ RUN      ] testSort.sortElementListDescending
[       OK ] testSort.sortElementListDescending (2 ms)
[ RUN      ] testSort.sortOneElementList
[       OK ] testSort.sortOneElementList (1 ms)
[ RUN      ] testSort.sortEmptyList
[       OK ] testSort.sortEmptyList (1 ms)
[----------] 4 tests from testSort (7 ms total)

[----------] Global test environment tear-down
[==========] 18 tests from 4 test cases ran. (134 ms total)
[  PASSED  ] 18 tests.
==12639==
==12639== HEAP SUMMARY:
==12639==     in use at exit: 0 bytes in 0 blocks
==12639==   total heap usage: 1,114 allocs, 1,114 frees, 98,203 bytes allocated
==12639==
==12639== All heap blocks were freed -- no leaks are possible
==12639==
==12639== For counts of detected and suppressed errors, rerun with: -v
==12639== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Izpis 4.7 Izpis rezultata naših enotskih testov brez puščanja pomnilnika v orodju *Valgrind*.

Uporabe orodja ne bi radi omejevali le na primer naše izvorne kode, pač pa bi radi raziskali kako dobro zaznava napake pri uporabi pomnilnika. To je privedlo do odločitve, da namerno pokvarimo našo izvorno kodo skozi serijo različnih testov in preverimo uspešnost detekcije teh napak. V sledečih razdelkih opišemo kako smo kodo okvarili in do kakšnih ugotovitev nas je privedlo to orodje.

4.2.2 Odstranitev operatorja `delete` pri enoti *remove*

V *C++* jeziku rezerviramo spomin na kopici (angl. *heap*) za en objekt s preprostim klicem `new`. Operatorju podamo tip objekta, ki ga želimo imeti na kopici, ta pa nam

vrne kazalec na naslov, kjer se nahaja novo nastali objekt. Primer uporabe v našem programu najdemo v enoti "append", in sicer vsakič, ko želimo ustvariti novo vozlišče, uporabimo `Node<T> *n = new Node<T>(val, NULL)`. Preko kazalca kasneje v programu manevriramo z objektom. Komplement operatorja `new` je `delete`, namenjen za sprostitev alociranega pomnilnika. V enoti "remove" vsakič poiščemo podani element, nastavimo kazalec prejšnjega elementa na naslednjega (če obstaja) in z operatorjem `delete` sprostimo spomin, ki ga je zasedal iskani element. Če pomnilnika ob brisanju ne dealociramo, ostane objekt še vedno v spominu, čeprav noben element ne kaže nanj. V primeru, da v sistemu z velikim pomnilnikom ustvarimo majhno število objektov na kopici, uporabnik najbrž ne bi opazil razlike v delovanju. Če pa tekom programa alociramo ogromne količine pomnilnika in jih nikoli ne sprostimo, lahko operacijski sistem sčasoma zavrne sistemski klic za dodeljevanje pomnilnika in ne moremo več kreirati novih objektov na kopici, ali pa nam lahko celo zamrzne (angl. *freeze*) in odpove celoten sistem. Veliko je odvisno tukaj tudi od operacijskega sistema, in sicer od tega kako in kdaj procesu odvzema resurse.

Za prvi test smo v enoti "remove" na dveh mestih zakomentirali operator `delete` za brisanje glave seznama in ostalih elementov ter opazovali reakcijo orodja. Enotskih testih nismo testirali, ker je v ozadju *Google Test* ogrodja zapletena arhitektura in je v primeru napake tako težje razbrati iz *Valgrind* izpisa, kje tiči problem. V `main` funkciji najprej ustvarimo seznam in mu dodamo tri elemente, npr. *3*, *4* in *5* v tem zaporedju. Nato jih odstranimo v obratnem vrstnem redu kot smo jih dodali, torej *5*, *4* in *3*. Pričakujemo, da se bo program izvedel brez napak, a ne bo sprostil pomnilnika za odstranjene elemente. Program ponovno prevedemo, ker smo spremenili `main` funkcijo (ne poganjamo več enotskih testov) in ga poženemo z ukazom `valgrind --tool==memcheck --leak-check=full ./runtests`. Izpis 4.8 prikazuje, da se program pravilno izvede, a nas ob koncu *Valgrind* obvesti o treh napakah pri ravnanju s pomnilnikom in o *48* definitivno izgubljenih (angl. *definitely lost*) bajtih. Orodje nam za vsak uporabljen `new`, ki ga nismo sprostili, z `delete` izpiše v kateri vrstici enote "append" smo ga uporabili in kje v funkciji `main` opravimo klic na enoto "append". V *LEAK SUMMARY* še enkrat izpiše koliko bajtov nam je še ostalo nesproščenih, v *ERROR SUMMARY* pa koliko pomnilniških napak imamo.

```
==5835== Memcheck, a memory error detector
==5835== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==5835== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
```

```
==5835== Command: ./runtests
==5835==
==5835==
==5835== HEAP SUMMARY:
==5835==     in use at exit: 48 bytes in 3 blocks
==5835==   total heap usage: 296 allocs, 293 frees, 42,005 bytes allocated
==5835==
==5835== 16 bytes in 1 blocks are definitely lost in loss record 1 of 3
==5835==    at 0x4C2B0E0: operator new(unsigned long)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5835==    by 0x450284: LinkedList<int>::append(int) (LinkedList.h:28)
==5835==    by 0x448A43: main (main.cpp:15)
==5835==
==5835== 16 bytes in 1 blocks are definitely lost in loss record 2 of 3
==5835==    at 0x4C2B0E0: operator new(unsigned long)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5835==    by 0x450284: LinkedList<int>::append(int) (LinkedList.h:28)
==5835==    by 0x448A50: main (main.cpp:16)
==5835==
==5835== 16 bytes in 1 blocks are definitely lost in loss record 3 of 3
==5835==    at 0x4C2B0E0: operator new(unsigned long)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5835==    by 0x450284: LinkedList<int>::append(int) (LinkedList.h:28)
==5835==    by 0x448A5D: main (main.cpp:17)
==5835==
==5835== LEAK SUMMARY:
==5835==    definitely lost: 48 bytes in 3 blocks
==5835==    indirectly lost: 0 bytes in 0 blocks
==5835==      possibly lost: 0 bytes in 0 blocks
==5835==    still reachable: 0 bytes in 0 blocks
==5835==         suppressed: 0 bytes in 0 blocks
==5835==
==5835== For counts of detected and suppressed errors, rerun with: -v
==5835== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Izpis 4.8 Izpis ob odstranitvi `delete` operatorja.

Zanimivo je tudi preveriti ali izpis orodja ostane isti, če spremenimo vrstni red odstranjevanja vozlišč. Ponovno dodamo elemente *3*, *4* in *5*. Tokrat jih odstranimo v istem zaporedju, kot smo jih dodali. Iz izpisa 4.9 je razvidno, da se je poročilo spremenilo. Tokrat je precej krajše in nas obvesti samo za en klic `new` operatorja. Prav tako je tudi drugačna porazdelitev izgubljenih bajtov, 16 jih je definitivno izgubljenih, 32 pa je izgubljenih indirektno (angl. *indirectly lost*), kar pomeni, da puščamo pomnilnik v strukturi, ki bazira na kazalcih [22]. Najprej odstranimo glavo seznama *3*, zato to orodje zabeleži to kot definitivno izgubljene bajte, ko pa odstranimo naslednika *4* in *5*, se ta dva kvalificirata pod indirektno izgubljenimi bajti, ker so bila vozlišča med seboj pove-

zana. *Valgrind* predpostavlja, da če rešimo najprej problem direktno izgubljenih bajtov, se bomo posledično rešili tudi indirektno izgubljenih, kar v našem primeru drži [22].

```
==5938== Memcheck, a memory error detector
==5938== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==5938== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==5938== Command: ./runtests
==5938==
==5938==
==5938== HEAP SUMMARY:
==5938==     in use at exit: 48 bytes in 3 blocks
==5938==   total heap usage: 296 allocs, 293 frees, 42,005 bytes allocated
==5938==
==5938== 48 (16 direct, 32 indirect) bytes in 1 blocks are
         definitely lost in loss record 3 of 3
==5938==    at 0x4C2B0E0: operator new(unsigned long)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5938==    by 0x450284: LinkedList<int>::append(int) (LinkedList.h:28)
==5938==    by 0x448A43: main (main.cpp:15)
==5938==
==5938== LEAK SUMMARY:
==5938==    definitely lost: 16 bytes in 1 blocks
==5938==    indirectly lost: 32 bytes in 2 blocks
==5938==      possibly lost: 0 bytes in 0 blocks
==5938==    still reachable: 0 bytes in 0 blocks
==5938==         suppressed: 0 bytes in 0 blocks
==5938==
==5938== For counts of detected and suppressed errors, rerun with: -v
==5938== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Izpis 4.9 Izpis odstranjevanja v obratni smeri pri izbrisu `delete`.

Z okvarjeno kodo lahko ponovno poženemo tudi enotske teste v *Google Test* ogrodju. Ker smo okvarili samo enoto "remove", lahko s stikalom `--gtest_filter=testRemove` določimo, da ogrodje zažene samo testne primere, ki testirajo to enoto. Rezultat testiranja je uspešen kljub puščanju pomnilnika, orodje pa nam ne namiguje na morebitne težave z dealokacijo izbrisanih vozlišč. To potrjuje, da se *Google Test* ne zmeni za preverjanje pomnilnika.

4.2.3 Odstranitev destruktorja

Destruktor je pomemben konstrukt *C++* jezika. Njegovo ime nam pove, da je komplement konstruktorja. Konstruktor kličemo ročno ob instanciranju objekta, medtem, ko se destruktor kliče avtomatsko ob uničenju objekta. Pri *C++* je večkrat omenjen tudi programerski idiom *Resource Acquisition Is Initialization* (RAII), ki poudarja pomembnost

dealokacije v destruktorju. Pomembnost tega načela je opazna predvsem pri izjemah, kjer se lahko program predhodno zaključi in za seboj pusti sledi nesproščenega pomnilnika, če smo imeli funkcijske klice za dealokacijo na koncu programa in ne v destruktorju objekta. V našem primeru smo ustvarili seznam vedno na skladu (angl. *stack*), vozle seznama pa na kopici (angl. *heap*). Ko se program zaključi in se kreiran seznam odstrani iz sklada, se avtomatsko pokliče njegov destruktor. Tukaj je pomembno, da sprostimo vsa preostala vozlišča v seznamu.

Za destruktor smo v programu poskrbeli že pri prvotnem testiranju z *Google Testom*, a smo ga zaradi namena *Valgrind* testiranja izbrisali. Pričakujemo, da bo orodje zaznalo pomanjkanje destruktorja in nas obvestilo o nesproščenem pomnilniku ob zaključku programa. Pri testiranju smo uporabljali le enoto "append", ki rezervira pomnilnik za vozlišča, sprosti pa ga ne. Na sklad potisnemo seznam z inicializacijo `LinkedList<int> list` in mu pripnemo pet poljubnih števil. Zaženemo program v okolju orodja in preverimo rezultat ob zaključku programa. Iz izpisa 4.10 lahko razberemo, da naš program ob zaključku pusti sled petih nesproščenih blokov. Podobno kot pri prejšnjem testu, nam orodje tudi tukaj izpiše vrstico prvega uporabljenega `new` operatorja, ki ga nismo izbrisali z `delete`. Ponovno se razdeli 16 bajtov v defintivno izgubljenih in preostalih 64 v indirektno izgubljenih, ker orodje prepozna sorodnost pomnilniškega problema.

```
==6405== Memcheck, a memory error detector
==6405== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6405== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==6405== Command: ./runtests
==6405==
==6405==
==6405== HEAP SUMMARY:
==6405==     in use at exit: 80 bytes in 5 blocks
==6405==   total heap usage: 298 allocs, 293 frees, 42,037 bytes allocated
==6405==
==6405== 80 (16 direct, 64 indirect) bytes in 1 blocks are
            definitely lost in loss record 5 of 5
==6405==    at 0x4C2B0E0: operator new(unsigned long)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==6405==    by 0x44F374: LinkedList<int>::append(int) (LinkedList.h:28)
==6405==    by 0x448A01: main (main.cpp:15)
==6405==
==6405== LEAK SUMMARY:
==6405==    definitely lost: 16 bytes in 1 blocks
==6405==    indirectly lost: 64 bytes in 4 blocks
==6405==      possibly lost: 0 bytes in 0 blocks
==6405==    still reachable: 0 bytes in 0 blocks
```

```
==6405==          suppressed: 0 bytes in 0 blocks
==6405==
==6405== For counts of detected and suppressed errors, rerun with: -v
==6405== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Izpis 4.10 Poročilo pri odstranitvi destruktorja.

Od orodja seveda ne moremo pričakovati, da nas točno obvesti o pomanjkanju destruktorja, ker ne more vedeti zakaj uporabljamo strukturo kazalacev. Prav tako je destruktor le ena od možnih rešitev pri sproščanju pomnilnika in je njegova uporaba odvisna od implementacije rešitve.

4.2.4 Uporaba ukaza `delete[]` namesto `delete`

Operator `new` torej uporabljamo za instanciranje enega objekta. Večkrat pa se zgodi, da bi radi hranili več objektov skupaj v tabeli (angl. *array*). Takrat uporabimo `new[]`, kjer znotraj oglatih oklepajev povemo velikost tabele. Pomnilnika alociranega z `new[]` ne sproščamo več z `delete`, pač pa z `delete[]`. S tem prevajalniku nakažemo, da gre za objekt tabelarne oblike. Tako se programerji večkrat zmotimo in za sproščanje pomnilnika uporabimo napačen operator. Pri tem testu smo v destruktorju zamenjali `delete` z `delete[]` in od orodja pričakujemo, da nas bo točno opozoril na napačno uporabo `delete[]`. Na izpisu 4.11 je vidno, da je orodje detektiralo napačen operator in poudarilo kritično vrstico v destruktorju. Izpostavilo je, kateri operator smo uporabljali za alokacijo in katerega za dealokacijo ter opozorilo na neujemanje (angl. *mismatch*). Opazimo lahko, da nam kljub napaki pomnilnik ne pušča, a je napačna uporaba operatorjev močno odsvetovana, ker povzroča nedefinirano obnašanje (angl. *undefined behaviour*) programa. Podoben test lahko naredimo še s *C* funkcijo `free`. Veliko programerjev, ki so vajeni *C* jezika in naredijo prehod na *C++* se lahko zaradi stare navade zmoti pri dealokaciji in tako uporabi `free` namesto `delete`. Iz izpisa 4.12 vidimo, da *Valgrind* zazna tudi to napako in nam pokaže, kje smo se zmotili.

```
==6520== Memcheck, a memory error detector
==6520== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6520== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==6520== Command: ./runtests
==6520==
==6520== Mismatched free() / delete / delete []
==6520==    at 0x4C2C83C: operator delete[](void*)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==6520==    by 0x44F94B: LinkedList<int>::~LinkedList() (LinkedList.h:22)
==6520==    by 0x448A9E: main (main.cpp:19)
```

```
==6520==  Address 0x5c494d0 is 0 bytes inside a block of size 16 alloc'd
==6520==    at 0x4C2B0E0: operator new(unsigned long)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==6520==    by 0x450FE4: LinkedList<int>::append(int) (LinkedList.h:28)
==6520==    by 0x448A62: main (main.cpp:15)
==6520==
==6520==
==6520== HEAP SUMMARY:
==6520==     in use at exit: 0 bytes in 0 blocks
==6520==   total heap usage: 298 allocs, 298 frees, 42,037 bytes allocated
==6520==
==6520== All heap blocks were freed -- no leaks are possible
==6520==
==6520== For counts of detected and suppressed errors, rerun with: -v
==6520== ERROR SUMMARY: 5 errors from 1 contexts (suppressed: 0 from 0)
```

Izpis 4.11 Poročilo pri uporabi `delete[]`.

```
==6719== Memcheck, a memory error detector
==6719== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==6719== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==6719== Command: ./runtests
==6719==
==6719== Mismatched free() / delete / delete []
==6719==    at 0x4C2BDEC: free
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==6719==    by 0x448A53: ~LinkedList (LinkedList.h:22)
==6719==    by 0x448A53: main (main.cpp:19)
==6719==  Address 0x5c494d0 is 0 bytes inside a block of size 16 alloc'd
==6719==    at 0x4C2B0E0: operator new(unsigned long)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==6719==    by 0x450254: LinkedList<int>::append(int) (LinkedList.h:28)
==6719==    by 0x448A03: main (main.cpp:15)
==6719==
==6719==
==6719== HEAP SUMMARY:
==6719==     in use at exit: 0 bytes in 0 blocks
==6719==   total heap usage: 298 allocs, 298 frees, 42,037 bytes allocated
==6719==
==6719== All heap blocks were freed -- no leaks are possible
==6719==
==6719== For counts of detected and suppressed errors, rerun with: -v
==6719== ERROR SUMMARY: 5 errors from 1 contexts (suppressed: 0 from 0)
```

Izpis 4.12 Poročilo pri uporabi `free`.

4.2.5 Uporaba ukaza `delete` na že izbrisanem vozlu

Pogosta programerska napaka pri upravljanju s pomnilnikom je brisanje že izbrisanega. Lahko se torej zgodi, da za isti kazalec na objekt dvakrat pokličemo `delete`. To ne pomeni dvakratno varnost proti puščanju pomnilnika pač pa nedefinirano obnašanje, največkrat sesutje programa med izvajanjem (angl. *runtime crash*). Pred testiranjem okvarimo izvorno kodo z dvakratno uporabo `delete(node)` v destruktorju. Zanima nas, ali bo to napako orodje zaznalo in kakšno bo poročilo. Na izpisu 4.13 prvo opazimo *Invalid free() / delete / delete[] / realloc()*, ki nakazuje na napako pri uporabi ene izmed naštetih opcij za sproščanje pomnilnika. Orodje nam že takoj izpostavi problematičen `delete` in njegovo vrstico. Opozori nas, da poskušamo sprostiti 0 bajtov pomnilnika oz. kazalec na vozlišče, katerega smo enkrat že dealocirali. Za lažje iskanje napake nam ponudi tudi vrstico prve uporabe `delete`, s čimer lažje preverimo, kateri je pravilen. To je zelo koristna informacija, sploh če se operatorja `delete` nahajata na dveh različnih mestih v kodi in je njuno podvojenost težje opaziti.

```
==8818== Memcheck, a memory error detector
==8818== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8818== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==8818== Command: ./runtests
==8818==
==8818== Invalid free() / delete / delete[] / realloc()
==8818==    at 0x4C2C2BC: operator delete(void*)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8818==    by 0x44FAE6: LinkedList<int>::~LinkedList() (LinkedList.h:24)
==8818==    by 0x448A9E: main (main.cpp:19)
==8818==  Address 0x5c494d0 is 0 bytes inside a block of size 16 free'd
==8818==    at 0x4C2C2BC: operator delete(void*)
            (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8818==    by 0x44FADE: LinkedList<int>::~LinkedList() (LinkedList.h:23)
==8818==    by 0x448A9E: main (main.cpp:19)
==8818==
==8818==
==8818== HEAP SUMMARY:
==8818==     in use at exit: 0 bytes in 0 blocks
==8818==   total heap usage: 298 allocs, 303 frees, 42,037 bytes allocated
==8818==
==8818== All heap blocks were freed -- no leaks are possible
==8818==
==8818== For counts of detected and suppressed errors, rerun with: -v
==8818== ERROR SUMMARY: 5 errors from 1 contexts (suppressed: 0 from 0)
```

Izpis 4.13 Poročilo pri dvakratni uporabi `delete`.

4.2.6 Uporaba ukaza `delete` za neobstoječ vozel

Daljši niz objektov običajno vedno spravimo v neke vrste tabelo, v našem primeru v seznam. Tabele objektov so skozi izvajanje programa spremenljive dolžine, zato čez njih iteriramo z zankami. Če se hočemo sprehoditi čez celotno tabelo, začnemo pri prvemu elementu in zanko ponavljamo, dokler ne dosežemo zadnjega. Kot smo že omenili, smo prvi element v našem seznamu poimenovali *head* in zadnji *tail*. Med njima se lahko nahaja poljubno število elementov, zato uporabimo `while(node)`, ki v skrajšani obliki ponavlja zanko toliko časa, dokler obstaja vozel oz. kazalec na vozel ni *NULL*. Kaj pa, če bi za iteracijo v zanki uporabljali določene meje in bi se pri določanju teh mej zmotili ter prekoračili seznam? Za zadnji test smo poskušali izbrisati vozel izven mej seznama. V destruktorju torej namesto `while(node)` uporabimo `for (int i = 0; i < (_size + 1); i++)` in nalašč prestopimo meje seznama za en element. Nato znotraj zanke, kot prej, sprostimo vsako vozlišče v seznamu. Ko pridemo do zadnje iteracije, pričakujemo, da bo orodje zaznalo, da posegamo po pomnilniku izven našega seznama in nas bo o tem informiralo. Na izpisu 4.14 vidimo, da se je program v okviru orodja zaključil s signalom *SIGSEGV*, katerega je poslal operacijski sistem, ker smo posegali po nedovoljeni lokaciji v pomnilniku. Orodje je ukaz sproščanja na neobstoječem vozlu vseeno izvršilo, nas je pa z stavkom *Access not within mapped region* opozorilo, da smo prestopili meje seznama v pomnilniku. Obvesti nas tudi, da naslova, ki smo ga poskušali sprostiti, nismo alocirali in pripomni, da je do tega mogoče prišlo zaradi prevelikega števila objektov na skladu (angl. *stack overflow*). Zanimivo je, da se nobeno vozlišče v seznamu ni sprostilo, čeprav je do napake prišlo šele na koncu seznama. Razlog take napake najbrž tiči v operacijskem sistemu, ki se je prej pritožil nad ilegalno operacijo v pomnilniku, kot je izvedel dejansko dealokacijo. Zaradi narave napake in implementacije sistema je nemogoče, da bi orodje kljub izvedbi kode preprečilo tako reakcijo. Na koncu lahko še vidimo, da je zaradi nenavadne napake kvalificiralo izgubljene bajte kot *possibly lost*.

```
==8506== Memcheck, a memory error detector
==8506== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8506== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==8506== Command: ./runtests
==8506==
==8506== Invalid read of size 8
==8506==    at 0x44FCB6: LinkedList<int>::~LinkedList() (LinkedList.h:19)
==8506==    by 0x448A9E: main (main.cpp:19)
==8506==  Address 0x8 is not stack'd, malloc'd or (recently) free'd
==8506==
```

```
==8506==
==8506== Process terminating with default action of signal 11 (SIGSEGV)
==8506==  Access not within mapped region at address 0x8
==8506==    at 0x44FCB6: LinkedList<int>::~LinkedList() (LinkedList.h:19)
==8506==    by 0x448A9E: main (main.cpp:19)
==8506==  If you believe this happened as a result of a stack
==8506==  overflow in your program's main thread (unlikely but
==8506==  possible), you can try to increase the size of the
==8506==  main thread stack using the --main-stacksize= flag.
==8506==  The main thread stack size used in this run was 8388608.
==8506==
==8506== HEAP SUMMARY:
==8506==     in use at exit: 8,534 bytes in 119 blocks
==8506==   total heap usage: 298 allocs, 179 frees, 42,037 bytes allocated
==8506==
==8506== LEAK SUMMARY:
==8506==    definitely lost: 0 bytes in 0 blocks
==8506==    indirectly lost: 0 bytes in 0 blocks
==8506==      possibly lost: 3,182 bytes in 62 blocks
==8506==    still reachable: 5,352 bytes in 57 blocks
==8506==         suppressed: 0 bytes in 0 blocks
==8506== Rerun with --leak-check=full to see details of leaked memory
==8506==
==8506== For counts of detected and suppressed errors, rerun with: -v
==8506== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

Izpis 4.14 Poročilo pri uporabi `delete` na neobstoječem vozlu.

4.2.7 Ugotovitve

Po izvedbi osnovnih pomnilniških napak in njihovih testov lahko potegnemo zaključek, da je *Valgrind* izvrstno orodje, ki nam lahko prikrajša dolge ure razhroščevanja. Izvajanje program, ki vsebuje pomnilniško napako, se lahko na naključnih mestih ustavi, zato je ročno določanje napake oteženo. To orodje pa nam konec vsakega zagona ustvari informativno poročilo, ki poskuša čimbolj opisati izvor problema, ali pa potrdi pravilno upravljanje s pomnilnikom. Vsak program in njegovo izvedbo z lahkoto testiramo za pomnilniško puščanje, zato je to orodje priporočljivo za vse razvijalce v jeziku *C++*.

5 Zaključek

V diplomskem delu smo skozi praktičen primer vzorčne izvorne kode poskušali prikazati koristi in relevantnost testiranja. Spoznali smo, da na nek način testiranje izvaja vsak razvijalec, a je za učinkovit potek testiranja in prepričljive rezultate potrebno predhodno sestaviti testni plan in se držati nekaterih izmed opisanih metod. Vsekakor je testiranje obvezen in neizogiben del vsakega projekta, ne glede na velikost in zahtevnost.

Napisali smo enotske teste s pomočjo *Google Test* ogrodja in jih pognali, da bi preverili pravilnost vsake testirane enote v programu. Ugotovili smo, da nam orodje za naš minimalističen primer programske opreme ni popolnoma koristil. Razen dveh testnih primerov, so se vse enote programa izvedle pravilno, a nas suhoparen izpis poročila testiranja ni dodobra prepričal o pravilnosti programa. Orodju primanjkuje raznih metrik zmogljivosti, ki bi ocenile testirane enote. Vseeno smo ugotovili, da so enotski testi lahko še vedno izjemno koristni predvsem v fazi razvoja, kjer lahko za vsako vnešeno spremembo v izvorni kodi ponovno poženemo teste in preverimo ali so se prvotne funkcionalnosti še vedno ohranile.

Med implementacijo enosmernega seznama smo se posluževali tudi manipulacije s po-

mnilnikom in shranjevanja objektov na kopici, zato smo bili primorani program testirati tudi za puščanje pomnilnika. Spoznali smo moč orodja *Valgrind* skozi serijo različnih testov, kjer smo nalašč okvarili kodo. Orodje je z veliko natančnostjo izpostavilo napake in ponudilo informativno stanje pomnilnika ob zaključku programa. Vsekakor se je izkazalo za nepogrešljiv del razvojne opreme vsakega *C* ali *C++* razvijalca, ki v svoji programski opremi ročno upravlja s pomnilnikom.

Delo bi lahko nadaljevali s preučitvijo uporabe ostalih funkcionalnosti ogrodja *Google Test* in *Valgrind* in le-te testirali na kompleksnejši izvorni kodi. Dodatno bi se osredotočili na testiranje že obstoječe kode, ki je dostopna na spletu in s tem ocenili njeno kvaliteto, hkrati s primernostjo ogrodij za uporabo na kodi z večjo kompleksnostjo. Naše delo bi lahko dodatno razširili s preučitvijo ostalih orodij za enotsko testiranje tako v jeziku *C++* kot tudi v ostalih programskih jezikih. Kljub temu pa smatramo, da smo z opravljenim delom dosegli svoj cilj, tj. pokazati pomembnost testiranja programske opreme, veliko učinkovitost enotskih testov in uporabnost orodij *Google Test* in *Valgrind*.

Literatura

[1] M. Mraz, Zanesljivost programske opreme, Dosegljivo: http://lrss.fri.uni-lj.si/sl/teaching/zzrs/lectures/4_Programska_oprema.pdf, [Dostopano: 30. 6. 2016] (2015).

[2] Blogspot, What are Different Goals of Software Testing?, Dosegljivo: http://testingbasicinterviewquestions.blogspot.si/2012/03/what-are-different-goals-of-software.html, [Dostopano: 30. 5. 2016].

[3] A. M. J. Hass, Guide to Advanced Software Testing, Artech House Publishers, Norwood, MA, 2008.

[4] L. Copeland, A Practitioner's Guide to Software Test Design, Artech House Publishers, Norwood, MA, 2004.

[5] K. Prasad, Advantages and Disadvantages of Black Box and White Box Testing, Dosegljivo: http://creativetesters678.blogspot.si/2008/07/advantages-and-disadvantages-of-black.html, [Dostopano: 3. 6. 2016] (2008).

[6] T. Dogša, Verifikacija in validacija programske opreme, Tehniška fakulteta, Maribor, 1993.

[7] Wikipedia, Jenkins (software), Dosegljivo: https://en.wikipedia.org/wiki/Jenkins_(software), [Dostopano: 19. 6. 2016].

[8] ISTQB exam certification, What is Integration testing?, Dosegljivo: http://istqbexamcertification.com/what-is-integration-testing/, [Dostopano: 11. 6. 2016].

[9] P. Ammann, J. Offutt, Introduction to software testing, Cambridge University Press, New York, 2008.

[10] Wikipedia, Software testing, Dosegljivo: https://en.wikipedia.org/wiki/Software_testing, [Dostopano: 19. 6. 2016].

[11] S. Rollins, Linked Lists, Dosegljivo: http://www.cs.usfca.edu/~{}srollins/courses/cs112-f08/web/notes/linkedlists/ll2.gif, [Dostopano: 13. 7. 2016] (2007).

[12] B. Mehta, Test Cases Vs Test Scenarios – Which is Better?(My Experience), Dosegljivo: http://www.softwaretestinghelp.com/test-cases-vs-test-scenarios/, [Dostopano: 6. 7. 2016] (2016).

[13] Software Testing Class, What is difference between Test Cases vs Test Scenarios?, Dosegljivo: http://www.softwaretestingclass.com/what-is-difference-between-test-cases-vs-test-scenarios/, [Dostopano 6. 7. 2016].

[14] Takanen, Ari and OUSPG crew, Glossary of Vulnerability Testing Terminology, Dosegljivo: https://www.ee.oulu.fi/research/ouspg/Glossary, [Dostopano: 2. 7. 2016].

[15] G. D. Everett, R. J. McLeod, Software Testing, John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.

[16] user7610, C++ unit testing framework, Dosegljivo: http://softwarerecs.stackexchange.com/questions/27897/c-unit-testing-framework, [Dostopano: 23. 7. 2016].

[17] Google, Primer, Dosegljivo: https://github.com/google/googletest/blob/master/googletest/docs/Primer.md, [Dostopano: 14. 7. 2016].

[18] Valgrind, Dosegljivo: http://valgrind.org/, [Dostopano: 21. 8. 2016].

[19] Memcheck: a memory error detector, Dosegljivo: http://valgrind.org/docs/manual/mc-manual.html, [Dostopano: 21. 8. 2016].

[20] The Design and Implementation of Valgrind, Dosegljivo: http://valgrind.org/docs/manual/mc-tech-docs.html, [Dostopano: 21. 8. 2016].

[21] A. Vairavan, How does Valgrind work?, Dosegljivo: http://i.stack.imgur.com/8SGij.jpg, [Dostopano: 21. 8. 2016] (2014).

[22] Valgrind Frequently Asked Questions, Dosegljivo: http://valgrind.org/docs/manual/faq.html, [Dostopano: 21. 8. 2016].